NPS-EC-14-002

NAVAL POSTGRADUATE SCHOOL

MONTEREY, CALIFORNIA

PERFORMANCE OF HIGH-RELIABILITY SPACE-QUALIFIED PROCESSORS IMPLEMENTING SOFTWARE DEFINED RADIOS

by

Herschel H. Loomis, Jr., George W. Dinolt, and Frank E. Kragh

March 2014

Prepared for: Secretary of the Air Force

REPORT DOCUMENTATION PAGE	*Form Approved* OMB No. 0704-0188

1. REPORT DATE ***(DD-MM-YYYY)*** 31-03-2014	**2. REPORT TYPE** Technical Report	**3. DATES COVERED** ***(From-To)*** 01-01-2012 to 31-12-2013
4. TITLE AND SUBTITLE Performance of High-Reliability Space-Qualified Processors Implementing Software Defined Radios		**5a. CONTRACT NUMBER**
		5b. GRANT NUMBER MIPR #R448212
		5c. PROGRAM ELEMENT NUMBER
6. AUTHOR(S) Loomis, Herschel H., Jr.; Dinolt, George W.; Kragh, Frank E.		**5d. PROJECT NUMBER**
		5e. TASK NUMBER
		5f. WORK UNIT NUMBER
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) AND ADDRESS(ES) Naval Postgraduate School, Department of Electrical and Computer Engineering, 833 Dyer Road, Monterey, CA 93943-5121		**8. PERFORMING ORGANIZATION REPORT NUMBER** **NPS-EC-14-002**
9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES) SAF/FMBIB-AFOY PO Box 14200 Washington, DC 20044-4200		**10. SPONSOR/MONITOR'S ACRONYM(S)**
		11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION / AVAILABILITY STATEMENT
Approved for public release; distribution is unlimited

13. SUPPLEMENTARY NOTES

14. ABSTRACT

This report provides results of a study of the application to software-defined radios (SDR) of the Maestro 49-tile Radiation-Hard-by-Design multi-processor chip developed by Boeing Corporation for the U.S. Government using DARPA-developed radiation-hard chip technology. The heart of the pipeline SDR architecture is an implementation of single-precision floating-point pipeline FFT. The details of the software architecture to achieve the pipeline operation are presented. The performance of *N*-point FFTs for *N* = 128, 256, 512, 1024, and 2048 is reported as number of processor tiles is increased. Maximum FFT throughput achieved for a 2048-point FFT is 27 million samples per second when 20 of the 49 available tiles are used for separate FFT blocks, one tile is used for input data distribution, and one tile is used for output data collection. The performance of the complete SDR is projected based upon the FFT experiments.

15. SUBJECT TERMS

Fault-tolerant Processors, Radiation-Hard Processors, FFTW, Software Defined Radio, SDR, Space-Qualified Processors

16. SECURITY CLASSIFICATION OF:			**17. LIMITATION OF ABSTRACT**	**18. NUMBER OF PAGES**	**19a. NAME OF RESPONSIBLE PERSON**
a. REPORT	**b. ABSTRACT**	**c. THIS PAGE**			Herschel Loomis
Unclassified	Unclassified	Unclassified	UU	76	**19b. TELEPHONE NUMBER** ***(include area code)*** (831) 656-3214

Standard Form 298 (Rev. 8-98)
Prescribed by ANSI Std. Z39.18

The report entitled "*Performance of High-Reliability Space-Qualified Processors Implementing Software Defined Radios*" was prepared for and funded by the Secretary of the Air Force.

Further distribution of all or part of this report is authorized.

This report was prepared by:

Herschel H. Loomis, Jr.
Distinguished Professor

George W. Dinolt
Professor of the Practice

Frank E. Kragh
Associate Professor

Reviewed by:

R. Clark Robertson, Chairman
Electrical and Computer Engineering

Released by:

Jeffrey D. Paduan
Dean of Research

THIS PAGE INTENTIONALLY LEFT BLANK

ABSTRACT

This report provides results of a study of the application to software-defined radios (SDR) of the Maestro 49-tile Radiation-Hard-by-Design multi-processor chip developed by Boeing Corporation for the U.S. Government using DARPA-developed radiation-hard chip technology. The heart of the pipeline SDR architecture is an implementation of single-precision floating-point pipeline FFT. The details of the software architecture to achieve the pipeline operation are presented. The performance of N-point FFTs for N = 128, 256, 512, 1024, and 2048 is reported as number of processor tiles is increased. Maximum FFT throughput achieved for a 2048-point FFT is 27 million samples per second when 20 of the 49 available tiles are used for separate FFT blocks, one tile is used for input data distribution, and one tile is used for output data collection. The performance of the complete SDR is projected based upon the FFT experiments.

THIS PAGE INTENTIONALLY LEFT BLANK

TABLE OF CONTENTS

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF FIGURES

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF TABLES

THIS PAGE INTENTIONALLY LEFT BLANK

I. INTRODUCTION

A. PROBLEM STATEMENT

The work reported in this Technical Report was supported by the Secretary of the Air Force as a result of a proposal submitted in September of 2011. [1]

1. Specific Tasks Originally Proposed

- We will develop algorithms for demonstration of the Maestro chip in space-based [Software Defined Radio] SDR applications. We will produce MAESTRO-development-board based demonstration of fundamental SDR processes, such as fast Fourier transforms (FFTs), finite impulse response (FIR) filters, forward error corrections encoders and decoders, and synchronization algorithms. If a MAESTRO development board is unavailable, we will make use of our Tilera 64-core development board.
- We will investigate the application of our previously developed SDR application to the [Single Event Immune Reconfigurable FPGA] SIRF-developed RHBD Virtex-5 chip.

2. Structure of Funded Work

The proposed work was funded in two annual increments, distributing the original requested funding over the two years, FY 2012 and FY 2013. Period of performance of the FY2013 increment was extended to 12/31/13.

B. BACKGROUND

Software defined radios (SDR) consist of a radio frequency front end (RFFE), data converters (analog to digital and digital to analog converters), and reconfigurable digital hardware. The modulation and encoding (for a transmitter) and the demodulation and decoding (for a receiver) are performed by the reconfigurable digital hardware which can be a microprocessor, a digital signal processor (DSP), a field programmable gate array (FPGA), or some combination of these. Since the modulation and coding are determined by the program that the microprocessor or DSP runs or the program that configures the FPGA, these radios are called software defined radios (SDRs). SDRs are reprogrammable, allowing them to transmit and receive any communications signal, provided the RFFE can accommodate the frequencies involved, the data converter has sufficient sample rate and dynamic range, and the reconfigurable digital hardware has sufficient processing capability.

Radiation in space poses a considerable threat to modern microelectronic devices, in particular to the high-performance low-cost computing capability we enjoy on earth. These threats apply as well to the processors one would employ implementing SDRs in space. These effects can be categorized as long-term permanent faults called *total dose effects* and transient temporary effects called *single event upsets* (SEU). [2]

Total dose effects must be mitigated by semiconductor manufacturing process modification, or by selecting and testing parts to meet the total dose requirements of the planned mission. Single event upsets are more difficult to prevent in modern, high-speed, small-feature-size devices. So, while total-dose radiation-tolerant modern processors and FPGAs are available, most modern current generation processors are very susceptible to SEUs [2].

Two notable exceptions to this generalization have emerged recently. The U.S. Government has been sponsoring the OPERA project through which the Boeing Corp. has produced a Radiation-Hard by Design (RHBD) 49-core (tile)[1] multiprocessor chip (MAESTRO) based on the architecture of the Tilera Corp. 64-tile chip [3]. This chip is being made available to U.S. Government space computing applications. It has been demonstrated at a U.S. Government sponsored Industry Day 9/29-9/30/2010 [4]. The other is a project sponsored by the Air Force Research Lab for Xilinx Corp. to develop a RHBD version of its Virtex-5 Field Programmable Gate Array (FPGA) chip [5].

It has long been understood that replication of logic with voting circuitry can be used to improve the reliability of digital systems in the presence of transient errors in the logic, such as SEUs. [2] [6] We at the Naval Postgraduate School (NPS) have been engaged in a project to build an evaluation board for a Triple Modular Redundant (TMR) implementation of a RISC processor to validate the TMR architecture for employment in a high-SEU environment. This evaluation board has evolved to a dual-FPGA processor called the Configurable Fault-Tolerant Processor (CFTP). The research has led us to the conclusion that the TMR architecture is an effective one to enhance the resistance of a

[1] *Core* is the more commonly-used term for a processor on a multi-processor chip. Tilera Corp. uses the synonym *tile* in its literature and as part of its name.

processor to SEUs so that the computer could operate reliably in the hostile environment of low earth orbit. [7]

The NPS is conducting research and education programs in SDR, including thesis research in SDR design of transceivers for IEEE 802.11 wireless LANs, IEEE 802.16 wireless MANs, and IS-95B and cdma2000 mobile telephony, and the course EC4530 *Soft Radio*. This work includes software defined radios consistent with the Software Communications Architecture (SCA), microprocessor-based SDRs, and most recently FPGA-based SDR design. The Naval Postgraduate School's Communications Research Laboratory is equipped for SDR design with eight software defined radio design stations including programming design environments, RFFEs, and microprocessor and FPGA modules.

SDRs are a natural fit for satellite applications because they can be changed via reprogramming after launch, thereby allowing new functionality and/or design improvement at any time in the spacecraft's lifecycle. It is expected this will make the satellite more useful over its lifespan including more operationally responsive. Furthermore, a single SDR can receive multiple dissimilar communications signals simultaneously and be reconfigured to receive different signals at different times – for example, different signals over different areas of the world.

The Naval Postgraduate School is currently at work on a project to design the software for a fault tolerant SDR suitable for hardware (FPGAs) already on orbit. The proposed SDR will process pre-demodulated signals in order to compress the signals for potential passing to the downlink. It is presumed that the downlink does not have sufficient bandwidth to pass the entire pre-demodulated signal. The compression algorithm will be configurable by ground operators who will set signal power thresholds for frequency ranges and time durations of interest. The compression will be accomplished by passing only those frequency ranges-time durations of the signal that exceed the relevant power threshold. The basic SDR design has been proven by Wright [8] and further refined by Livingston [9]. The FPGA configuration is being made fault tolerant by applying the methods learned in this research program and will be tested on the Algorithmic WorkStation (AWS) prior to being tested on an on-orbit FPGA.

A key component of this SDR is a high-speed pipeline Fast Fourier Transform (FFT) unit. We have had an earlier research effort on the realization of high-speed, pipelined FFTs. It developed the architecture for a high-speed pipelined signal processor for the computation of the Cyclic Spectrum [10] of which the principal component is an FFT processor.

A recent thesis has developed the realization of a Radix-4 64-point real-time FFT implemented and simulated in a Virtex-II FPGA [11]. This design was implemented with both TMR and RPR fault-amelioration techniques and showed a modest improvement in resource utilization of the RPR technique over TMR. Unfortunately, the fault-tolerant FFTs were not available in time for testing last summer in the UC Davis cyclotron.

The NPS investigators also have experience with the multi-core processor architecture that is exploited in the Boeing-developed MAESTRO chip. We have investigated ways to enhance the designed-in RHBD technology of the chip [12]. We have looked into ways to utilize the multi-core architecture for the implementation of SDRs and some particular SIGINT algorithms.

The NPS investigators have also had some hands-on experience with the Tilera processor on which the MAESTRO is based. At the time we investigated how we might implement highly reliable, high-speed implementations of encryption and hashing algorithms utilizing the pipelined architecture available on the Tilera. We investigated how to take advantage of the allocations of specific portions of the chip to specific functions, i.e. how the chip design supports physical redundancy and where there might be potential single points of failure. We compared this architecture to the Cell Broadband Engine, a different multi-core approach. Although we did not do a complete implementation of the encryption algorithms on the Tilera, our analysis indicated that its speed for hashing and encryption would be roughly comparable that of the Cell with possibly greater resistance to hardware failure.

Based on this experience with the implementation of high-speed pipelined processors and the design of high-performance reliable processors for the space environment, we have studied the use of the Maestro RHBD multi-core processor for the implementation of a SDR to perform data compression on broad-band pre-demodulated signals.

C. REPORT ORGANIZATION

In Chapter II, the architecture of the pipelined SDR is developed and techniques for the implementation of the pipelined FFT on Maestro are developed. Chapter III presents the design of the Maestro-Development-Board hosted experiments and the results of the experiments. Finally, conclusions and recommendations for future research and development efforts are the topic of Chapter IV.

II. PIPELINED SDR ARCHITECTURE

A. SPECIFIC PRE-D DATA COMPRESSION SDR

The basic SDR that was the motivation and implementation target for the research was developed in master's theses by Livingston [9], Wright [8], and Humberd [13]. The radio would monitor a band of the RF spectrum of bandwidth *B*. It would convert that band-limited portion of the RF to digital samples at a sample rate, f_s, such that $f_s > 2B$, the Nyquist rate. Then, the SDR computes *N*-point FFTs of each successive *N*-point block of input data samples. For each *N*-point block of complex frequency data, the magnitudes of all the positive frequency components are computed. Then, the frequency indices of each magnitude that exceeds a specified threshold [and in a specified frequency sub-band] are identified and the corresponding complex-frequency values are reported. Figure 1 Illustrates how this works. The left-hand panel shows the magnitude of the FFT versus time. A block of five time-units worth of data is transformed at a time. The SDR is looking for significant signal components, i.e., data above the "blue-contour" magnitude, while ignoring weak signal components. The magnitudes exceed the threshold in the red-boxed areas in the spectrum diagram. The right-hand panel of the figure shows the frequency components of the compressed signal. Only those components in the red boxes are outputted, as frequency index and complex amplitude.

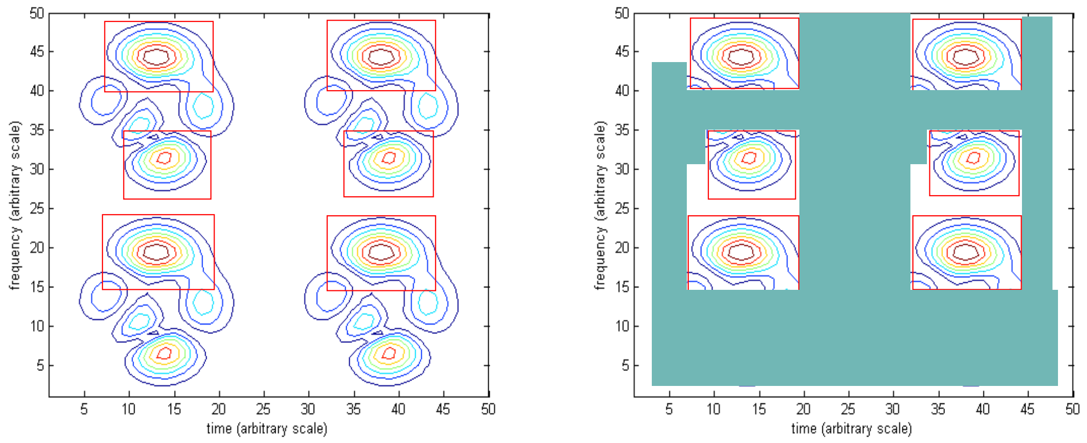


Figure 1. Basic Concept of SDR (After [8])

If this SDR were placed in a satellite, then the selected frequency components would be downlinked with a block identifier. Ground processing can reconstruct the significant components of the signal by performing the inverse FFT on the selected frequency components, block-by-block.

The frequency resolution of the SDR is simply *N*, the block size of the FFT, so the potential compression ratio will be *N/k* for blocks with *k* FFT indices with power greater than the threshold. For blocks with less signal power, no frequency components are downlinked achieving an infinite compression ratio, although probably null blocks should have their time stamp downlinked.

Figure 2 shows the block diagram of the computational processes that are required to implement the SDR. It is desired that these processes be implemented in real time with sample rates in the tens of megahertz. Two basic ways to accomplish this goal would be:

1. By use of a FPGA or Application-Specific Integrated Circuit (ASIC) implementing pipeline versions of the major processor sub-blocks of Figure 2.
2. By use of a multi-processor to compute separate *N*-point blocks of selected frequency components in parallel.

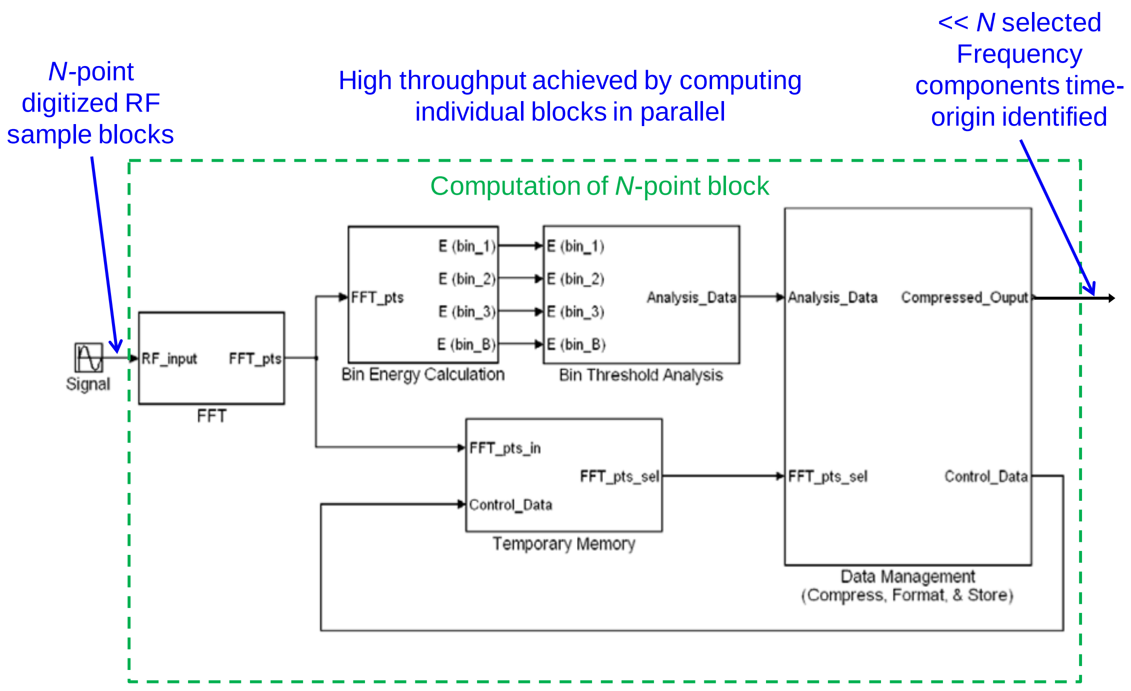


Figure 2. Block Diagram of Wright's SDR (After [8])

The use of parallel processors to do the computation relies on the fact that each *N*-point FFT is independent of the others. So, as long as the blocks are time stamped, the computation of Figure 2 can be carried out in a different processor with the selected frequency-component blocks with their time stamp reassembled at the output. This latter approach is the one that would be suitable for implementation of the SDR on a multi-core processor such as Maestro.

B. MAESTRO FFT ARCHITECTURES

The basic Maestro multi-core architecture is shown in Figure 3. The architecture of Maestro uses the intellectual property of the Tilera Corporation for its 64-core commercial architecture. This architecture was purchased by the U.S. Government for royalty-free use by the Government in space applications. The Boeing Corporation was contracted by the Government to produce a 49-core RHBD chip, incorporating the basic Tilera architecture and adding an IEEE-standard floating-point co-processor to each core.

Tilera refers to its *cores* as *tiles* so we will use the term *tile,* which corresponds to the usage in Tilera documentation.

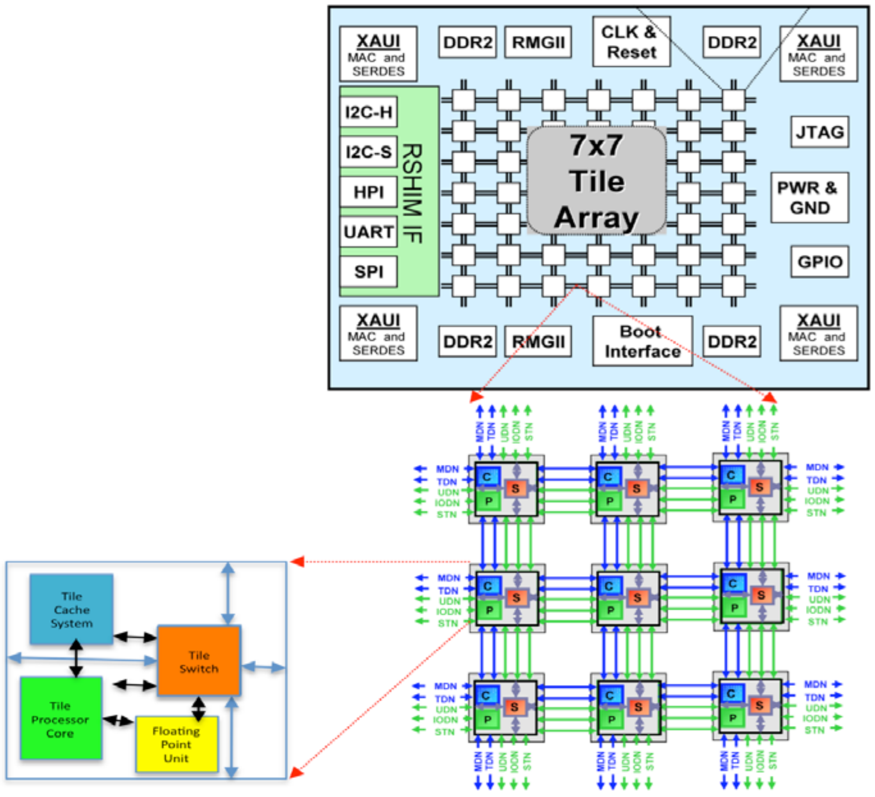
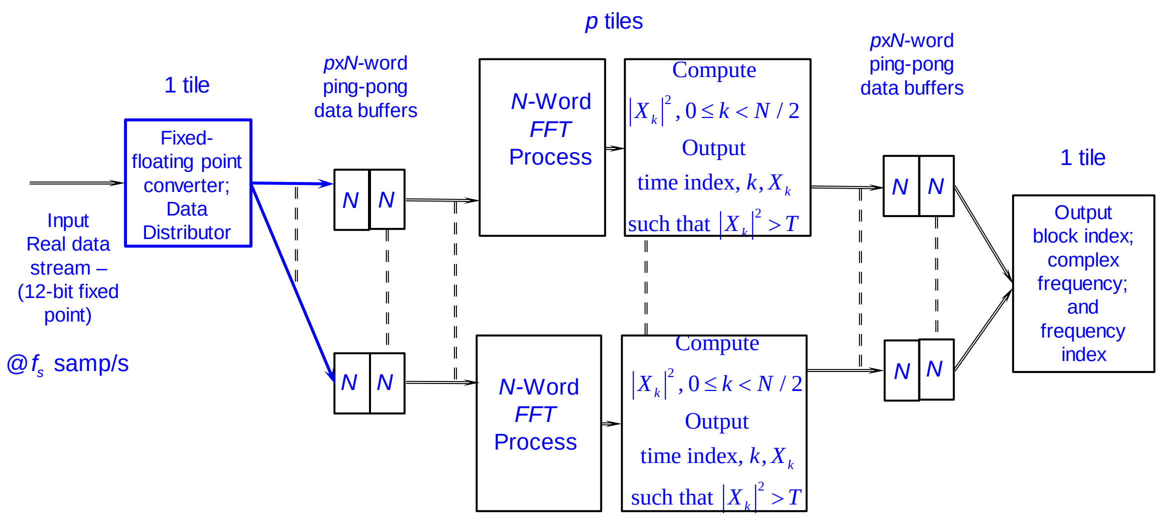


Figure 3. Maestro 49-core MIPS Processor Architecture [14].

1. Maestro SDR Architecture

Figure 4 shows the basic architecture of the planned multi-core architecture of the Maestro program to compute the real-time spectrum of the incoming sampled data stream, select the components whose power exceeds a given threshold, and then output the time index of the *N*-sample block and the frequency indices and complex magnitudes of the spectrum.



p tiles
1 tile
pxN-word ping-pong data buffers
Fixed-floating point converter; Data Distributor
Input Real data stream – (12-bit fixed point)
@f_s samp/s
N N
N-Word FFT Process
Compute $|X_k|^2, 0 \le k < N/2$ Output time index, k, X_k such that $|X_k|^2 > T$
pxN-word ping-pong data buffers
1 tile
Output block index; complex frequency; and frequency index

Figure 4. Real-Time Basic Pipeline FFT Architecture as Applied to SDR

The first tile in the process, the source tile, converts each 12-bit sample into a 32-bit IEEE standard floating-point number and places the samples into *p* successive *N*-word buffers. As each *N*-word buffer is filled, the source signals the associated FFT-select tile, "Ready to Send a Block."

Each of those *p* tiles performs the following operations:

- It waits for that "ready-to-send" signal and when received, initiates a direct-memory-access (DMA) transfer of the block of data with time stamp into the empty half of the input ping-pong buffer.
- It checks if the output ping-pong buffer is available and if it is,
    - Computes the FFT of the full half of the input ping-pong buffer;
    - tests the magnitude of the power in each positive-frequency component of the spectrum;
    - Loads the time stamp, number of components exceeding threshold, frequency indices and their complex amplitudes into the empty half of the output ping-pong buffer;
    - Signals the output is available to the output tile.

The sink tile then performs the following operations:

- It waits for a signal from any of the *p* FFT-compute tiles, "ready to send";
- It initiates a DMA transfer of the data block from that tile;
- It sends that data off chip.

Programming of the Maestro chip to exploit the parallelism displayed in Figure 4 is a very difficult process. The programmer must explicitly manage the data transfer DMA operations between tiles, manage the ping-pong data buffering, as well as provide computationally efficient processes to compute the FFTs and test for the significant frequency components. Because of this complexity, it was decided to first implement a simplified parallel algorithm to develop the techniques for distributing the data and to exploit powerful FFT algorithms developed by others.

2. FFT Program Used for FFT Tiles

The FFT program used in the tests reported here is a version of the FFTW ("Fastest Fourier Transform in the West") algorithm reported by Singh, et al. [14] In that paper, the authors describe their adaptation of the FFTW algorithm to the Maestro chip and their simulation studies of the performances with various sizes of FFT on Maestro and their extrapolation of multi-tile performance. They calculated a net Floating Point Operations (FLOPs) per clock cycle for a variety on FFT sizes. These results are summarized in Table 1.

Table 1. FLOPs per Clock Cycle for Various FFT Sizes (after [14])

FFT size	64	256	1024	2048	4096
Flops/Cycle [14]	0.51	0.5	0.33	0.41	0.35
FLOPs/FFT	1920	10,240	51,200	112,640	245,760
Single-tile f_s	5.95×10^6	4.38×10^6	2.31×10^6	2.61×10^6	2.04×10^6
p-Tile f_s	$5.95\times10^6\,p$	$4.38\times10^6\,p$	$2.31\times10^6\,p$	$2.61\times10^6\,p$	$2.04\times10^6\,p$

We analyzed the results from [14] to obtain a digital sample rate or throughput for a multi-tile FFTW implementation on Maestro. Based upon a number of real FLOPs per FFT of $N\log_2 N$, we estimate a single-tile sample rate achievable by their FFTW also shown in Table 1. Assuming no data distribution overhead in the operation of *p* FFT tiles in parallel operating on different blocks, the sample rate should scale linearly with *p*, as shown in the final row of Table 1. This final estimate gives us an upper bound on the sample rate achievable from a *p*-tile parallel pipeline implementation of the ISI FFTW in accordance with the structure shown in Figure 5.

This upper bound suggests that a 20-tile pipelined FFT could achieve real-time FFT operating at a sample rate of 52 Mega-samples per second or less.

In the next chapter, we discuss the details of our experiment to obtain a real-time pipeline FFT and to verify the operation of our pipeline SDR architecture. Finally, it presents the results of the performance experiments.

III. MAESTRO-BASED FFT EXPERIMENTS

A. PIPELINE FFT IMPLEMENTATIONS

The process illustrated in Figure 4 that the FFT-select tile performs has two basic components, the calculation of an *N*-point floating-point FFT and the selection of the frequency components to downlink. The FFT has a computational requirement of $C_{FFT} = 5N\log_2 N$ real floating point operations, whereas the selection portion of the algorithm has $C_s = 3\frac{N}{2}$ flops plus $\frac{N}{2}$ integer comparisons. (See Section D. for further discussion of these complexity figures.) The dominant computational requirement comes from the FFT, and hence it was decided that the most important process to implement would be the multi-tile pipeline FFT. The structure of that pipeline real-time FFT process is shown in Figure 5. This is very similar to the pipeline SDR architecture shown in Figure 4; the selection portion of each tile's process has been removed.

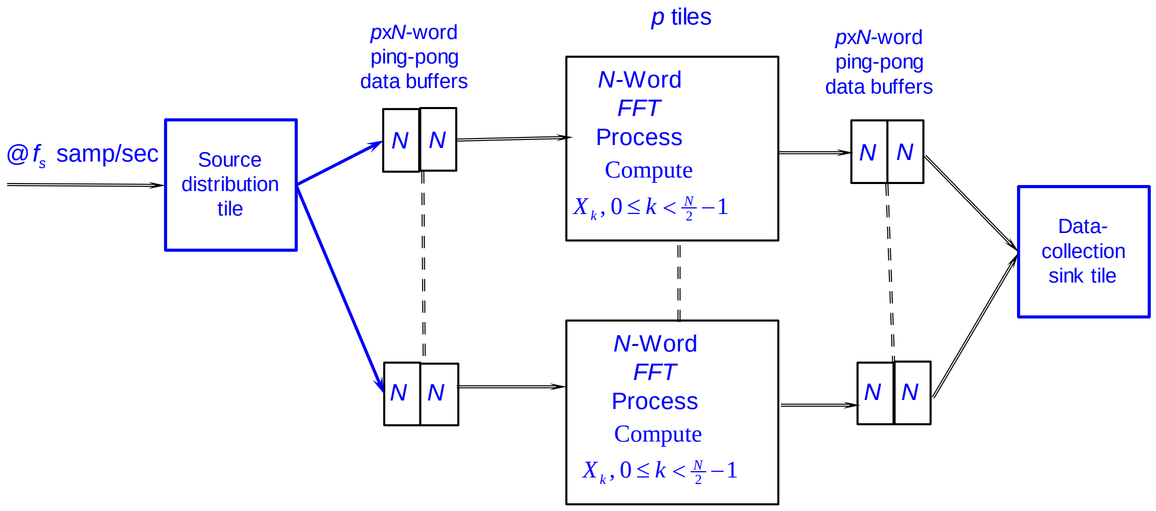

@f_s samp/sec
Source distribution tile
pxN-word ping-pong data buffers
N
N
N
N
p tiles
N-Word FFT Process
Compute
$X_k, 0 \le k < \frac{N}{2} - 1$
N-Word FFT Process
Compute
$X_k, 0 \le k < \frac{N}{2} - 1$
pxN-word ping-pong data buffers
N
N
N
N
Data-collection sink tile

Figure 5. Real-Time Pipeline FFT Architecture

The operation of the pipeline FFT architecture is very similar to that of the SDR architecture; only the selection process has been eliminated. The first tile in the process, the source tile, converts each 12-bit sample into a 32-bit IEEE standard floating-point number and places the samples into *p* successive *N*-word buffers. As each *N*-word buffer is filled, the source signals the associated FFT-select tile, "Ready to Send a Block."

We used the system interfaces to the underlying tile-to-tile communications functions provided in the MAESTRO "*ilib*" library and the "tmc" library functions to allocate memory shared among processes. Documentation for these libraries is distributed as part of the MAESTRO development environment [15].

Note that each FFT tile is executing a separate Unix process with its own "memory address space." Along with a significant amount of book keeping and error detection, each of these processes does the following:

- Receive Parameters from source process. This includes the address of the shared memory buffers used to transmit blocks from the sender to the receiver. We use the *ilib_msg_broadcast* library call to receive this message
- Allocate "ping-pong" buffers using *tmc_cmem_memalign* to share with the "data collection" process.
- Send a message to the data collection process via the *ilib_send_msg* call of the address of the shared memory
- Receive message from "source" via the *ilib_receive_msg* call that a message is ready to be collected.
- Copy via Direct Memory Access (DMA) the first source block from the source using the *ilib_mem_start_dma* call and wait via the *ilib_wait* call for the DMA to complete the copy. The *ilib_mem_start_dma* call sets up internal structures on the two associated tiles and uses separate mechanisms for the copy to take place. The CPU is not directly involved in the copy process and can do other computations while the copying is going on. When the copy is complete internal registers are set and the CPU will wait for that to happen via the *ilib_wait* call.
- Start loop $p - 1$ times (p, number of parallel tiles, passed from the source process)
    - Receive message from source that another source buffer is ready and then start DMA copy into 2nd buffer, but do not *wait.*
    - Process FFT on the first buffer while the DMA copy is taking place using the *fftwf_execute_dft_r2c* call.

    - Send a message to the data collection sink that this buffer is ready using *ilib_send_msg*
    - Using *ilib_wait*, wait for the DMA started above to complete
- Process the last FFT block and let data collection sink know.

The source and sink processes operate in a similar fashion using the same calls. There is a 4th process, that starts each of the source, FFTW and sink tile processes and establishes which specific tiles each runs on. This process "spawns" these by filling in a set of parameters that describe the process to be run (via its file name), the number of instances of the process and the location of the instances, passing this parameter to the *ilib_proc_spawn* library call.

Note that the ordering of the *N*-sample complex-frequency-amplitude blocks is maintained by the inclusion of the time stamp. This will permit recreation of the band-limited sampled signal by simply taking the inverse FFT of each block.

Next, we implemented the structure of Figure 5 to experimentally measure the sample rate of the parallel FFTW tiles. The C code for the FFTW was obtained from the Maestro source code distribution web site. [15] The single-precision FFTW was only able to be compiled without optimization. A version of a single tile's code was compiled for each value of block size (*N*) tested. The code used had a separately-compiled "wisdom" file, used for FFTW internal optimization, for each block size tested, so that FFTW code would not spend time setting up its configuration. The binary code for each tile's FFTW, including the ping-pong buffers is approximately eight MBytes. In the object code generated, each floating-point instruction appears to be padded by 4-5 no-ops. The reason for this apparently has to do with the communications between the floating point co-processor and the main CPU. Each floating point instruction takes more time than the completion of the message between the two entities.

1. Verification of the Correctness of the FFTW

The single-tile FFTW compiled code was tested for functional correctness for values of *N* that would be used in the pipeline multi-tile performance tests, namely for $N \in \{128, 256, 512, 1024, 2048\}$. For each value of *N*, a number of random data blocks were generated and submitted to the compiled FFTW code. Those results were compared to the results of Matlab® FFTs computed on the same random data blocks but using double precision. The results agreed within the least significant mantissa bit of our single-precision output. As a result, we had confidence that the compiled FFTW code was functionally correct and that the performance data would be for a functionally correct FFTW.

B. FFT PERFORMANCE MEASURING EXPERIMENTS

The experiments were conducted on the Maestro Development Board, loaned to the Naval Postgraduate School by the U.S. Government. The board was operating at a clock frequency of 350 MHz.

The software to implement the architecture of Figure 5 was created, compiled and loaded on the MDB and 100,000 *N*-word blocks were submitted to the various programs. The average throughput was measured and is reported for each value of *N* and for the various numbers of parallel FFT-computing tiles. The samples are each 32-bit IEEE standard floating point words. The pseudo-code statement of the experiment structure is shown in Table 2.

Table 2. Pseudo-code for Maestro FFT Performance Test

```
$f = 350,000,000$
for $\boxed{N \in \{128, 256, 512, 1024, 2048\}}$
    for $p = 1:20$
        for $R = 1:20$
            process 100,000 $N$-sample FP FFTs
            count Maestro cycles, $C(N,p,R)$
        end for
        compute average $\boxed{\overline{C}(N,p) = \frac{1}{20}\sum_{R=1}^{20} C(N,p,R)}$
        compute $\boxed{\overline{S}(N,p) = \frac{f \times N \times 100,000}{\overline{C}(N,p)} \left( \frac{\text{samples}(\text{cycles}/\text{sec})}{\text{cycles}} \right)}$
    end for
end for
plot family of curves for $\boxed{\overline{S}(N,p) \text{ vs. } p}$
```

where *N* is FFT block size, *p* is the number of FFT tiles, and *R* is the number of runs.

C. FFT EXPERIMENTAL RESULTS

The raw data, $\overline{S}(N,p)$, collected from the experiments is given in Appendix C.

The results from the experiment are shown in Figure 6. In that figure are plotted the curves of pipeline sample rate or throughput in millions of samples per second versus number of FFT tiles for each of the five values of *N*.

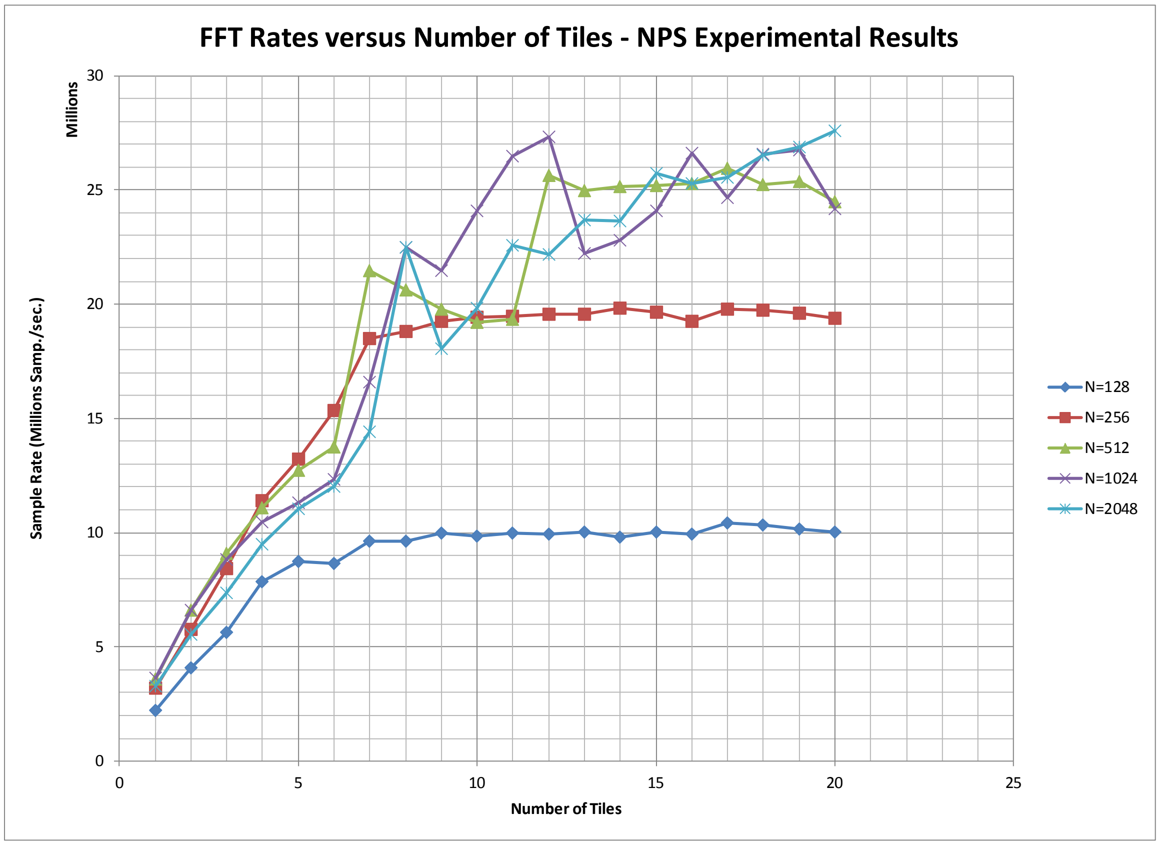


Figure 6. Chart of NPS Experimental Results

Discussion:

- At lower block sizes, 256 and 512, it appears that at some point the cost setting up the DMA (*ilib_dma_start_dma)* and processing the wait for termination (*ilib_wait)* are dominating the processing time, so that even though the number of tiles increases, the cost of the DMA overwhelms the potential benefit of the additional FFT tiles. DMA setup and wait cost is most likely independent of the size of the data being moved.

- At larger block sizes we see some increase in performance with increasing number of tiles beyond the 7 or 8 with 256 and 512 size blocks, but in these cases it appears that eventually we are seeing contention on the various internal buses. Part of the reason we think that is because of the variability that we are seeing. This may also be influenced by the arrangements we used for the tiles. With Figure 3 as a reference, the tiles are labeled (x,y) where $0 \leq x, y \leq 6$. The source tile was placed at (1,3) and the sink tile at (1,4). The up to 20 FFT tiles were placed in the columns starting at (2,0) with the tiles used in a block of height 7 and width 3, through tile (4,6). We let the library (os) determine which tiles within a block were used for a particular count of FFT tiles. We did not experiment by setting up different arrangements of tiles.

Earlier, we discussed the results from the ISI paper [14] that appear to set an upper bound on the performance of the pipelined multi-tile implementation of the FFTW. The ISI projected performance of two of the FFT sizes that coincide with sizes that were used in the NPS experiments compared to the NPS results for the same FFT sizes are shown in Figure 7.

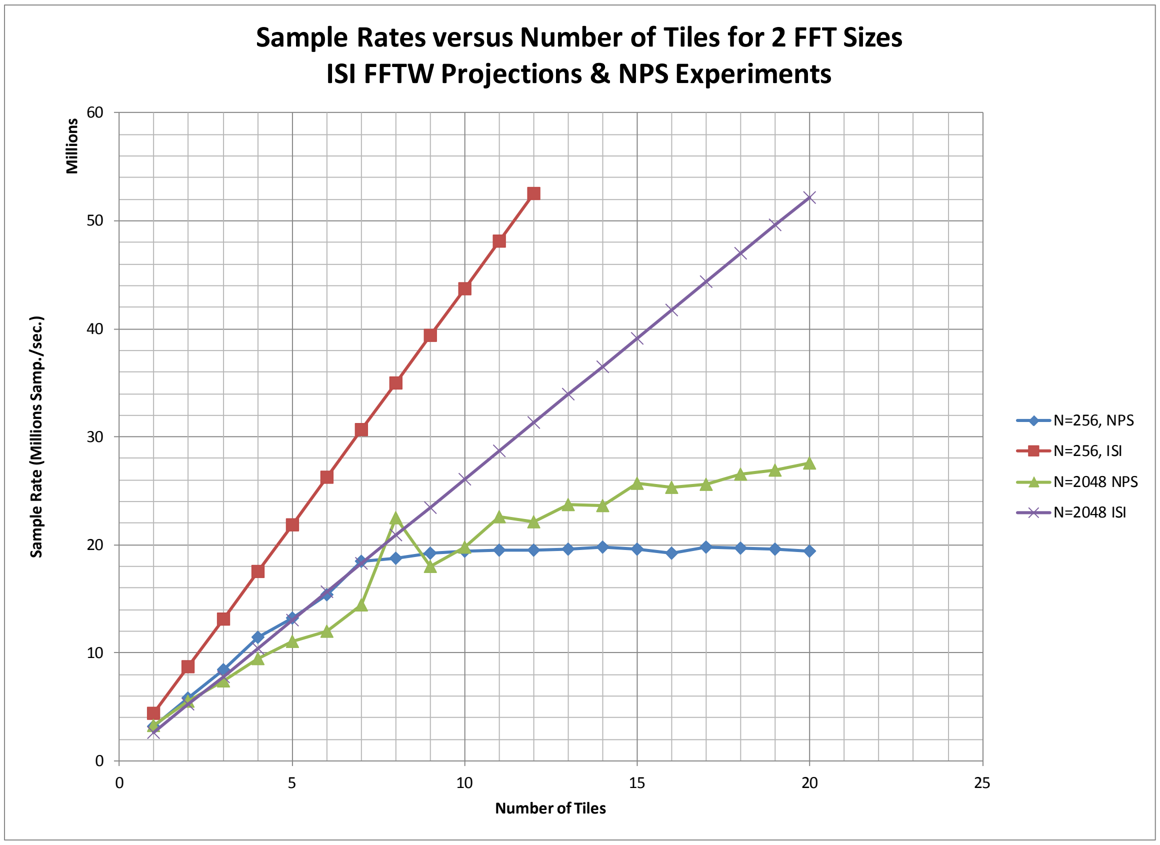


Figure 7. Projections from ISI paper compared to NPS results.

For *N* = 256, the NPS experimental results significantly underperform the ISI projections. We believe this to be caused by the DMA overhead. For *N* = 2048 and for low tile numbers, the NPS performance is close to the upper bound, until it reaches the knee of the NPS curve and then internal bus contention appears to take over.

D. APPLICATION OF RESULTS TO SDR PERFORMANCE

The process of programming the Maestro Development Board (MDB) to investigate the performance of the multi-tile FFTW was much more difficult than we had estimated at the outset of the project. Furthermore, a working MDB was only obtained in September of 2013. Consequently experimental verification of the SDR performance was not obtained.

Nevertheless, we can make reasonable predictions of the SDR performance from our FFT experiments.

Referring back to the basic pipeline architecture of Figure 4, we see that in addition to computing an *N*-point FFT, each of the *p*-processing tiles must compute

$\frac{N}{2}+1$ values of $\left|X_k\right|^2$ and test each value with respect to a real threshold[2]. Each tile will then output via DMA *q* 16-bit frequency indices and 2*q* 32-bit floating point complex frequency real or imaginary parts, where $0 \le q \le \frac{N}{2}-1$.

The computational requirements of the *N*-point FFT is $5N\log_2 N$ FLOPs. The computational requirements of the magnitude squaring and testing are $3\frac{N}{2}$ FLOPs and $\frac{N}{2}$ integer[3] operations. Thus, the computational requirements for each processing tile will increase from $5N\log_2 N$ FLOPs to $5N\log_2 N + 2N$, a modest increase indeed. Consequently, when the number of tiles in Figure 6 exceeds ten (the knee of those curves) and the performance is limited by DMA performance and bus contention, it is expected that the throughput for the full SDR implementation will be nearly the same as for the FFT alone. Furthermore, since the SDR is basically a data compression process, the output data rate should be much less than *N* 32-bit words per block, allowing a further modest increase in throughput.

[2] Since the signal being processed is real, only the positive portion of the frequency spectrum need be tested and downlinked.

[3] Since IEEE floating point numbers are normalized, comparisons of floating-point numbers can be performed using 32-bit integer arithmetic.

IV. CONCLUSIONS AND RECOMMENDATIONS FOR FURTHER STUDY

A. CONCLUSIONS

- A pipeline parallel multi-tile IEEE Single Precision Floating Point version of the FFTW was coded and tested on a Maestro Development Board with from 1 to 20 parallel tiles computing FFTs with block sizes of 256 through 2048.
- Pipeline throughputs for these FFTs were achieved of up to 25 million 32-bit samples per second.
- Addition of the rest of the SDR code to each FFT tile should not decrease performance for number of tiles, $p > 10$ and for $N > 512$.
- Higher block size operates more efficiently.
- The pipeline architecture was successfully demonstrated.
- Programming a single application to exploit parallelism of a multi-tile processor like Maestro is very difficult.
    - Because the caches are relatively small, main memory access is relatively expensive and inter-tile communications is not super fast, one has to take care in explicitly managing memory and inter-tile communications. The tools to do this are available, to some extent, but take some understanding. In our case, we are unsure whether the “main loop” of the FFT algorithm fit into the cache. We would need to do more evaluation to determine this.
    - We depended on the compiler to take advantage of the potential built-in instruction parallelism. Except in a couple of cases, we were unable to exploit the very long instruction word parallelism directly ourselves.
    - The system provides a set of development tools and libraries. The current compiler has some problems, especially handling the optimization of single precision floating-point arithmetic. Although the libraries are documented and there are tools for evaluating and optimizing code, understanding when to use which features of the

system takes some experience. In addition, it is unclear how the use of the various features might interact with each other.

B. RECOMMENDATIONS FOR FUTURE STUDY

The ISI paper [14] and Table 1 suggest that an upper bound on throughput of $2.6 \times 10^6 p$ samples per second might be achieved for each tile computing a 2048-point FFTW running under the pipeline architecture demonstrated in this study. This would mean that a 31-tile realization of the full SDR would potentially support a throughput of 80 million samples per second, enabling in-space processing of 32-MHz bandwidth signals. The following things should be tried to seek to realize that potential

- Verify the effect on throughput of adding the SDR functionality directly to the FFT tiles. Compare the performance of the alternative of adding a SDR tile in tandem to each FFT tile via the `run_sink.c` code.
- Experiment with different mapping of functionality to physical tiles.
- Work with ISI to optimize the NPS-developed code.

We have considered several approaches we could take to possibly optimize the FFT computations on the MAESTRO board. These include, listed in order of difficulty to address, but not necessarily the order of expected improvement:

1. Compiling/Coding Optimizations;
2. Dedicated Tiles;
3. Geometric Optimizations;
4. Refactoring the FFT Algorithms.

Below we describe these in more detail.

We think that the most improvement would come from some combination of Dedicated Tiles and Refactoring the FFT Algorithm.

FFT Algorithm. The only change we made to the delivered FFTW package is to recompile it for single precision floating point operation. This provided a small performance improvement. We have not spent any significant time and efforts applying the techniques outlined in the "Optimization Guide, UG105" [15] document to the FFTW package or our integration code. We would like to apply the various monitoring tools to analyze the performance of the FFTW package to see where the bottlenecks are. It would be interesting to apply the analysis tools to the package, apply the compiler feedback-based optimizations and add appropriate compiler features to the code. We are currently using the "`ilib`" interfaces for inter-tile communications and synchronization. There are "intrinsic" level compiler macros that directly access hardware level instructions to accomplish the communications and synchronization functions. These would remove the `ilib` function call overhead. It is not clear how much this would save but it is worth looking at and might provide for tighter (less latency) inter-tile communications.

Dedicated Tiles. We are currently using the delivered version of the Linux operating system. This version of the OS does not include any configuration of "dedicated tiles." We would like to configure and compile a new version of the operating system that includes dedicated tiles to do the FFT computations. We think this may provide significant performance improvements since the dedicated tiles operate with much less OS overhead than normal tiles, hence we may see more effective use of both the processor and cache.

Geometric Optimizations. We have only done a limited number of experiments on the allocation of our processes to tiles. Our current efforts do not show a significant increase in overall throughput, samples per second, once the number of tiles doing FFT processing increases beyond around 14 or so. We are not sure why this is the case, since our tile-to-tile communications speed measurements indicate that we should be seeing better performance than that. We think part of the problem is how the assignment of function to specific tiles is done. The length of the path between two communicating tiles increases latency and there is a potential for collisions on a network path where several tiles attempt to communicate over the same path simultaneously. Since we potentially know all the communications among processes, we should be able to find optimal, or at least better, arrangements of processes to tiles. If we were to build a new OS version, this could tie into where we allocate the dedicated tiles.

Refactoring The FFT Algorithm. The FFTW implementation of the FFT algorithm is configured to compute an FFT block on a single tile. We achieve our "throughput" by providing multiple tiles, each computing a full FFT block. Our current measurements indicate that we could come close to doubling the throughput if we could speed up the FFT computations. In that case the limiting factor would be the inter-tile communications. One way of achieving an increase in FFT processing speed is to refactor the algorithm to take advantage of the MIPS/MAESTRO capabilities. One approach is to "role our own" FFT implementation that still uses one tile/block but does not include any of the overhead needed to support the generality of the FFTW implementation. It can be tailored with appropriate assembly code to take advantage of the single precision floating point operations needed. This code could be tailored for exactly the block size we use and integrated into the inter-tile communications infrastructure. We could apply the various analysis and optimization techniques directly to this code. It is unclear, without further analysis, whether the implementation for, say, a 1K block size would fit comfortably on a single tile and cache. A second approach might be to attempt to factor the algorithm to run on multiple tiles to take advantage of the "butterfly" computations. This might increase the latency slightly but might make it easier to ensure that the computations take place entirely within the cache.

APPENDIX A

This appendix contains the source code for the various tile processes that were used in the experiment. These files are:

1. `Makefile:` Contains instructions for building the 4 programs: `startdma or start3dma, run_sender, run_receiver and run_sink.`
2. `simpledma.h: "Include file" that contains definitions of various macros and data types used throughout the programs.`
3. `startdma:` is the main driver for FFT testing. From command line arguments it determines how many FFT tiles to set up (run_receiver), assigns the tiles to each of the programs to be run and exits. Typical command format is:
   `startdma blocksize iterations numReceivers`
   where `blocksize` is the number of samples in one FFT block, one of (256, 512,1024, 2048)
   `iterations` is the number of blocks each "receiver" will process, typically a value above 100000 will give relative consistent results, and
   `numReceivers` is the number of tiles used to process FFTs, 1<= `numReceivers` <= 20
4. `run_sender`: is the program that generates data to be processed, it sends a dma message to a run_receiver to tell it that it has a buffer to process.
5. `run_receiver`: DMAs blocks from run_sender, FFTs the blocks, and signals `run_sink` that it has a block to process.
6. `run_sink`: DMAs blocks from `run_receiver` for future processing (not done here)
7. `start3dma`: is a simplified version of `startdma` that has only one `run_receiver` tile. It places `run_sender` on tile (1,3), `run_receiver` on tile (2,3) and `run_sink` on tile (3,3) This was constructed for testing.

1. makefile

```
/* Written by George W. Dinolt, Naval Postgraduate School, Monterey, CA
   Last Change Rev: 120, Last Changed Date: Fri, 22 Nov. 2013 */

      ifndef TILERA_ROOT
      $(error The 'TILERA_ROOT' environment variable is not set.)
      endif

      BIN = $(TILERA_ROOT)/bin/
      FFTW_ROOT = $(TILERA_ROOT)/src/tools/opera-fftw
      LIB_ROOT = $(TILERA_ROOT)/src/tools/opera-fftw/lib

      CONVERTFEEDBACK = ${BIN}/tile-convert-feedback

      CC = $(BIN)tile-cc
      CFLAGS =  -O2
```

```
LDFLAGS = -static -Os -L$(TILERA_ROOT)/tile/lib -
L$(TILERA_ROOT)/tile/usr/lib -L$(LIB_ROOT) -lfftw3f -lilib -ltmc
-lm

TARGETS = startdma startudma start3dma run_sender run_receiver
run_sink

all: ${TARGETS}

%:%.o
      ${CC} $< -o $@ ${LDFLAGS}

run_sender.o: run_sender.c simpledma.h

run_receiver.o: run_receiver.c simpledma.h

run_sink.o: run_sink.c simpledma.h

startdma.o: startdma.c simpledma.h

startudma.o: startudma.c simpledma.h

start3dma.o: start3dma.c simpledma.h

clean:
      rm -rf *.o ${TARGETS}
```

2. Include File – simpledma.h

```
/* Written by George W. Dinolt, Naval Postgraduate School, Monterey, CA
   Last Change Rev: 120, Last Changed Date: Fri, 22 Nov. 2013 */

      #ifndef __SIMPLEDMA
      #define __SIMPLEDMA


      #include <ilib.h>
      #include <stdio.h>

      #include <tmc/cmem.h>
      #include <arch/cycle.h>
      #include <sys/time.h>
      #include <fftw3.h>


      //sending buffers between sender and receivers
      #define MESSAGE_TAG 17
      #define MESSAGE_SENT_TAG 18

      //sending buffers between receivers and sink
      #define MESSAGE_3_TAG 19
      #define MESSAGE_3_SENT_TAG 20

      // sending last receipt
      #define MESSAGE_LAST_RECEIVED 21
```

```
// Ranks of various processes
#define SEND_RANK 0
#define SINK_RANK 1
#define FIRST_RECEIVER 2

// Size of shared memory objecta.
// blocksize is an input parameter

#define BUFFERSIZE(blocksize) (size_t)(blocksize * sizeof(float))

/*
 * Macros to "simplify" message and dma code by assuming
 *
 * ILIB_GROUP_SIBLINGS is always the group
 *
 * automagically converts variables into pointers
 * and calculates sizes where necessary
 *
 * assumes that "status" is defined where necessary
 */

//ILIB msg send
//Assume default group
#define SEND_MESSAGE(receiver, tag, buffer) \
  ilib_msg_send(ILIB_GROUP_SIBLINGS,          \
            receiver,            \
            tag,                 \
            &buffer,             \
            sizeof(buffer))

// ILIB msg receive
// assume status out and error message
#define RECEIVE_MESSAGE(sender, tag, buffer)\
  ilib_msg_receive(ILIB_GROUP_SIBLINGS,       \
               sender,           \
               tag,                    \
               &buffer,          \
               sizeof(buffer),         \
               &status)

#define BROADCAST_MESSAGE(source, buffer)\
  ilib_msg_broadcast(ILIB_GROUP_SIBLINGS,\
                 source,            \
                 &buffer,           \
                 sizeof(buffer),    \
                 &status)

#define DMA_START(destination_ptr, source_ptr, nbytes, request)
\
  ilib_mem_start_dma(destination_ptr,                       \
                 source_ptr,                    \
                 nbytes,                              \
                 &request)


typedef struct params {
  int blocksize;
  int buffersize;
  int iters;
  float *buffer1;
  float *buffer2;
  int nreceivers;
} params_t;


typedef struct args {
  int blocksize;
  int iters;
  int nreceivers;
} args_t;

#endif
```

3. startdma.c

```
/* Written by George W. Dinolt, Naval Postgraduate School, Monterey, CA
   Last Change Rev: 120, Last Changed Date: Fri, 22 Nov. 2013 */

   /*

   USAGE:

   ./startdma blocksize iterations numReceivers

   where:
         blocksize:    the sample size of the fft block to be computed,
                     one of 2^k for k in [5-11]

       iterations:   the number of ffts computed by each receiver,
                     at least one

       numReceivers: the number of receivers being run concurrently,
                     1<= numReceivers<=20

   OUTPUT is generated by the run_sender program and is sent to
   stdout. It is of the form:

   run_sender: iters=100000, nreceivers=1, BUFFERSIZE=2048
   run_sender: cycles = 4953052451
   run_sender: Cycles per block transfer = 49530
   run_sender:  Transfer Rate = 3617970 samples/sec

   where:
      iters is the number of FFTs each receiver did

      nreceivers is the number of receivers (between 1 andd 20)

      BUFFERSIZE is the size of the buffer needed to hold one block of
      FFT data

      cycles is the difference in the number of cycles reported at the
      receipt of the last ack from run_sink and the number at the
      beginning of the first send of an FFT as collected by
      get_cycle_count
      cycles/block transfer = cycles / ( iters * nreceiveers )

      samples/sec = (iters * nreceivers * (BUFFERSIZE / 4)  /
                    ( cycles * cyclesPerSecond)

                   which reduces to:

          (iters * nreceivers * BUFFERSIZE * cyclesPerSecond)/(4 *
cycles)

      This provides a consistent measure of sample rate for each
      blocksize and number of recievers.

   Driver Program: spawns:

   run_sender:  Program that generates packets for use

   run_receiver(s): Each receiver copies source blocks from sender and
   produces fft. There can be up to 20 receivers.

   run_sink: Program to receive all the fftw data. At reciept of last
   block it signals sender that it is done.

  See simpledma.h for descriptions of the macros:

  SEND_MESSAGE
  RECEIVE MESSAGE

  BROADCAST_MESSAGE

*/

#include "simpledma.h"

#define USAGE "USAGE: startdma blocksize iterations numReceivers"
#define UBLOCK(b)  b == 32 || b == 64 || b == 128 || b == 256 ||\
    b == 512 || b == 1024 || b == 2048

args_t *parseArgs(int argc, char *argv[])
{
  args_t * a = (args_t *)tmc_cmem_memalign((size_t)64, sizeof(args_t));

  if (argc != 4 )
    {
      ilib_die("startdma: wrong number of args, got %d\n\t%s",
             argc, USAGE);
    }
  a->blocksize = atoi(argv[1]);
  if ( ! (UBLOCK(a->blocksize)) )
    {
      ilib_die("startdma: blocksize=%d not power of 2\n\t%s", a-
>blocksize, USAGE);
    }
  a->iters = atoi(argv[2]);
  if ( a->iters <= 0 )
    {
      ilib_die("startdma: iterations=%s must be positive\n%s",
               argv[2], USAGE);
    }
  a->nreceivers = atoi(argv[3]);
  if ( a->nreceivers <= 0 || a->nreceivers > 20)
    {
      ilib_die("startdma: number of receivers = %s must b in range
1<=nr<20\n\t%s",
               argv[3], USAGE);
    }
  return a;
}

int main(int argc, char *argv[])
{
  ilib_init();
  args_t *a = parseArgs(argc, argv);

  const int nreceivers = a->nreceivers;

  ilibProcParam pparams[3]; // send, sink, receiver

  memset(pparams, 0, sizeof(pparams));

  /*
   * run_sender   at tile 1,2
   * run_sink     at tile 1,4
   * run_receiver at tiles [2-4],[0-6]

   */

  pparams[0].num_procs = 1;
  pparams[0].binary_name = "run_sender";
  pparams[0].init_block = a;
  pparams[0].init_size = sizeof(args_t);
  pparams[0].tiles.x = 1;
  pparams[0].tiles.y = 3;
  pparams[0].tiles.width = 1;
  pparams[0].tiles.height = 1;
  pparams[1].num_procs = 1;
  pparams[1].binary_name = "run_sink";
  pparams[1].tiles.x = 1;
  pparams[1].tiles.y = 4;
  pparams[1].tiles.width = 1;
  pparams[1].tiles.height = 1;
  pparams[2].num_procs = nreceivers;
  pparams[2].binary_name = "run_receiver";
  pparams[2].tiles.x = 2;
  pparams[2].tiles.y = 0;
  pparams[2].tiles.width = 3;
  pparams[2].tiles.height = 7;
  ilibGroup spawned_procs;
  if (ilib_proc_spawn(3, pparams, &spawned_procs) != ILIB_SUCCESS)
    {
      ilib_die("Failed to spawn.");
    }
  ilib_finish();
  return 0;
}
```

4. run_sender.c

```
/* Written by George W. Dinolt, Naval Postgraduate School, Monterey, CA
   Last Change Rev: 120, Last Changed Date: Fri, 22 Nov. 2013 */

/*
  Program to generate data for fftw speed testing.  It is
  expected that this program will be started via the startdma program
and assigned to specified tile.

  See simpledma.h for descriptions of the macros:

  SEND_MESSAGE
  RECEIVE MESSAGE
  DMA_START
  BROADCAST_MESSAGE

  General flow is,

  Get Parameters from command line argument set up in startdma using
  "ilib_prog_get_init_block"

  Initialize local buffers and local "Pointers"

  Initialize each block to be used with (single precision) floating
  point numbers

  set up and BROADCAST paramters for run_receiver and run_sink
processes

  get current cycle_count

  LOOP: (number of iterations)

      LOOP: over all run_receivers:
           SEND_MESSAGE next message to specified run_receiver that
            message is ready for DMA

  wait for last message from run_sink
  get current cyclce_count

  print out results.

define DEBUG for debugging


*/

#include "simpledma.h"


//#define DEBUG
```

```
#ifdef DEBUG
static void run_sender(args_t *a, FILE *rserr)

#else
static void run_sender(args_t *a)
#endif
{
  int i;
  int slot;
  int k;
  long long start_ctr, end_ctr, cycles;

  const int blocksize = a->blocksize;
  const int iters = a->iters;
  const int nreceivers = a->nreceivers;


#ifdef DEBUG
  fprintf(rserr, "Running sender with blocksize %d\n",blocksize);
  fflush(rserr);
#endif

  // Allocate a chunk of shared memory and fill it up.

  float *buffer1 = (float *) tmc_cmem_memalign((size_t)64,
BUFFERSIZE(blocksize));
  if ( buffer1 == NULL)
    {
      ilib_die("run_sender: Unable to allocate sender buffer1 memory");
    }

  float *buffer2 = (float *) tmc_cmem_memalign((size_t)64,
BUFFERSIZE(blocksize));
  if ( buffer1 == NULL)
    {
      ilib_die("run_sender: Unable to allocate sender buffer2 memory");
    }


  for (i = 0; i < blocksize; i++)
    {
      buffer1[i] = (float) i;
      buffer2[i] = (float) i;
    }

  float  *blocks[2];
  blocks[0] = buffer1;
  blocks[1] = buffer2;  // this should be "blocksize" floats


  params_t * p = (params_t *)tmc_cmem_memalign((size_t) 64,
sizeof(params_t));

  p->blocksize = blocksize;
  p->buffersize=BUFFERSIZE(blocksize);
  p->iters = iters;
  p->buffer1 = buffer1;
  p->buffer2 = buffer2;
  p->nreceivers = nreceivers;
  // Before allowing the receive to access the shared memory, we must
  // guarantee that all of the writes from the loop above have
  // completed.  Without this call, the receiving process might read a
  // stale value from the shared_memory array.

  // Send the memory address.

  ilibStatus status;
  BROADCAST_MESSAGE(SEND_RANK, p );
  if (status.error != ILIB_SUCCESS)
    {
      ilib_die("run_sender: unable to broadcast param message");
    }

#ifdef DEBUG
  fprintf(rserr, "run_sender: broadcast parameters address, %x\n", p);
  fflush(rserr);
#endif

  start_ctr = get_cycle_count();
  for ( i = 0; i< iters; i++)
    {
      slot = i % 2;
      blocks[slot][0] = i;
      blocks[slot][blocksize-1] = i;
      ilib_mem_fence();
      for (k = 0; k < nreceivers; k++)
      {
        SEND_MESSAGE(FIRST_RECEIVER + k, MESSAGE_SENT_TAG,i);

#ifdef DEBUG
        fprintf(rserr, "run_sender:  send buffer %d  to receiver %d
ready\n", i, k);
        fflush(rserr);
#endif
      }
    }

  long long rcv_end_ctr;
  RECEIVE_MESSAGE(SINK_RANK, MESSAGE_LAST_RECEIVED, rcv_end_ctr);

  end_ctr = get_cycle_count();
  cycles = end_ctr - start_ctr;
  printf("run_sender: iters=%d, nreceivers=%d, BUFFERSIZE=%d\n",
       p->iters, nreceivers, BUFFERSIZE(blocksize));
  printf("run_sender: cycles = %lld\n", cycles);
  long long cyclesPerBlock = cycles/(p->iters * nreceivers);
  printf("run_sender: Cycles per block transfer = %lld\n",
cyclesPerBlock);

  // do arithmetic this way to ensure no overflow

  // This is the number taken from
  // cat < /proc/cpuinfo
  // Unclear whether this is "real" or not
  const unsigned long long cyclesPerSecond = 350000000L;

  // divide by 4 to get samples (each sample is a real single precision
float
  const unsigned long long transferRate =
    ((unsigned long long)(p->iters) *
     (unsigned long long)nreceivers *
     (unsigned long long)BUFFERSIZE(blocksize) * cyclesPerSecond)/
    (unsigned long long)cycles/4;

  printf("run_sender:  Transfer Rate = %llu samples/sec\n",
         transferRate);

}

int main(void)
{
  ilib_init();
  int my_rank = ilib_group_rank(ILIB_GROUP_SIBLINGS);
  if ( my_rank != SEND_RANK)
    {
      ilib_die("send_sender:  rank = %d, wrong.", my_rank);
    }
  args_t *a;
  size_t a_size = 0;
  a = ilib_proc_get_init_block(&a_size);

#ifdef DEBUG
  FILE *rserr= fopen("run_sender_errors", "w");
  fprintf(rserr, "run_sender: Read a, got blocksize=%d, iters=%d,
nreceivers=%d\n",
          a->blocksize, a->iters, a->nreceivers);
  fflush(rserr);
#endif

  if (a_size != sizeof(args_t))
    {
      ilib_die("run_sender: size of args is wrong");
    }
#ifdef DEBUG
  run_sender(a,rserr);
#else
  run_sender(a);
#endif

  ilib_finish();
  return 0;
}
```

5. run_receiver.c

```
/* Written by George W. Dinolt, Naval Postgraduate School, Monterey, CA
   Last Change Rev: 120, Last Changed Date: Fri, 22 Nov. 2013 */

/*
```

```
  program to compute FFTW's reveived via DMA from run_sender.  It is
  expected that this program will be started via the startdma
  program. Multiple copies of this program may be started.  All copies
  dma blocks to process from the same source and make available fftw
  processed blocks for the single "sink" to dma blocks for post
  processing.

  See simpledma.h for descriptions of the macros:

  SEND_MESSAGE
  RECEIVE MESSAGE
  DMA_START
  BROADCAST_MESSAGE

  General flow is,

  Get Parameters from run_sender (BROADCAST_MESSAGE)

  Initialize local buffers and local "Pointers"

  Tell run_sink where the local buffer to copy from is

  get first received message from run_sender
  start DMA of buffer from run_sender
  wait for DMA to finish
  LOOP:
     RECEIVE_MESSAGE from run_sender that message is ready
     start DMA message to local memory
     fftw_previous buffer (while DMA is happening)
     SEND sink message that  buffer ready for DMA
     wait for started DMA to finish

DEBUG is for debugging
MEMCPY is for testing message passing without doing FFTW
PRINT_RECEIVER_OUT is for further debugging help.

*/

//#define MEMCPY

//#define DEBUG

//#define PRINT_RECEIVER_OUT
//#define MEMCPY

#include "simpledma.h"

/*
   fftw requires a "plan".  The plan depends on the "processor" and
some
   tests that fftw makes to determine the "best" approaches.  Plans
   can be saved, partially,  in "wisdom files".  If such a file is
   available then fftw can be instructed to use it. If    the Wisdeom
   file is not available, then fftw will generate a plan.  This takes
   some time, measured in seconds.

   Note that we are assuming that all the cores that will be doing
   fftw can use the same "wisdeom" file and same plan.  I have *not*
   tested that this is, in fact, the case.

   Indications are that buffer movement is a bigger problem.
*/

#ifndef MEMCPY
#define WISFILENAME 100
fftwf_plan setupWisdom(float *in, float *out, int blocksize)
{
  char wis[WISFILENAME];
  snprintf(wis,WISFILENAME, "Wisdomf_%d", blocksize);
  FILE *wfd = fopen(wis, "r");
  if (wfd != NULL)
  {
    fftwf_import_wisdom_from_file(wfd);
  }

  fftwf_plan p = fftwf_plan_dft_r2c_1d(blocksize, in,
                               (fftwf_complex *)out, FFTW_EXHAUSTIVE);

  if (wfd == NULL)
    {
      wfd = fopen(wis, "w");
      fftwf_export_wisdom_to_file(wfd);
      fclose(wfd);
    }

  return p;
}

#pragma frequency_hint INIT setupWisdom
#endif
void run_receiver(void)
{
  ilibStatus status;
  float *sink_blocks[2];

  int my_rank = ilib_group_rank(ILIB_GROUP_SIBLINGS);

  params_t *p;

#ifdef DEBUG
  fprintf(stderr, "run_receiver: rank=%d, waiting for param list\n",
my_rank);
  fflush(stderr);
#endif

  // Receive the params object address

  BROADCAST_MESSAGE(SEND_RANK, p);

  if (status.error != ILIB_SUCCESS ||
      status.size != sizeof(p)  )
  {
    ilib_die("run_receiver: Failed to receive params from sender");
  }
```

```
#ifdef DEBUG
  fprintf(stderr, "run_receiver: got params pointer %x\n", p);
  fflush(stderr);
#endif

  float  *buffer1 =  p->buffer1;     /* source  buffer address */
  float  *buffer2 =  p->buffer2;     /* source  buffer address */
  const int blocksize = p->blocksize;
  const int iters = p->iters;
  float  local_buf[2][blocksize];    /* destinatin buffers */
  float *src_blocks[2]; /* these are set to point to the source buffers
*/
  ilibRequest requests[2];

#ifdef DEBUG
  fprintf(stderr, "run_receiver: rank=%d, got buffer length of %d\n",
        my_rank, blocksize);
  fflush(stderr);
#endif

  // buffersize is size of 1 buffer (1024*sizeof(float), sink is double
buffered

  float *sink_buffer = (float *)tmc_cmem_memalign((size_t) 64, 2 * p-
>buffersize  );
  if (sink_buffer == NULL)
    {
      ilib_die("Unable to allocate transfer buffer\n");
    }

  // Tell sink about sink buffer

#ifdef DEBUG
  fprintf(stderr, "run_receiver: rank = %d, sending sink_buffer
address=%x to sink\n",
        my_rank, sink_buffer);  /* careful here */
  fflush(stderr);
#endif

  SEND_MESSAGE(SINK_RANK, MESSAGE_3_TAG, sink_buffer);

  // set address of src blocks, can only do this after buffer address
  // has been assigned.
  src_blocks[0] = buffer1;
  src_blocks[1] = buffer2;

  sink_blocks[0] = sink_buffer;
  sink_blocks[1] = sink_buffer + blocksize;

  int  i;
  int  j;
  int  slot;
  int  prevSlot;

#ifdef PRINT_RECEIVER_OUT
  long long start_ctr, end_ctr, cycles;
  start_ctr = get_cycle_count();
#endif

  slot = 0;

#ifdef DEBUG
  fprintf(stderr, "run_receiver: rank %d, waiting for first (0th)
message for sender\n",
        my_rank);
  fflush(stderr);
#endif

#ifndef MEMCPY
  fftwf_plan plan = setupWisdom(buffer1, sink_buffer, blocksize);
#endif

  RECEIVE_MESSAGE(SEND_RANK, MESSAGE_SENT_TAG, j);

  if (status.error != ILIB_SUCCESS ||
      status.size != sizeof(j) )
    {
      ilib_die("run_receiver: Failed ilib_msg_receive of buffer
ready");
    }

  if (j != 0)
    {
      ilib_die("Receiver got first message out of sequence\n");
    }

#ifdef DEBUG
  fprintf(stderr, "run_receiver: rank %d,  Starting first dma read
DMA\n",
        my_rank);
  fflush(stderr);
#endif

  if (DMA_START(local_buf[slot], src_blocks[slot],
BUFFERSIZE(blocksize), requests[slot])
      < 0)
    {
      ilib_die("Failed 1st DMA.");
    }

  if (ilib_wait(&requests[slot], &status) < 0)
    {
      ilib_die("run_receiver: failed on first message receive");
    }

  int prev;
  i = 0;
  while ( i < iters - 1 )
    {

#ifdef DEBUG
      fprintf(stderr, "run_receiver: rank=%d, expect %d,
src__blocks[slot] = %f\n,",
            my_rank, i, src_blocks[slot][0]);
      fflush(stderr);
#endif

      prev = i;
      prevSlot = slot;

      i++;
      slot = i % 2;

      RECEIVE_MESSAGE(SEND_RANK, MESSAGE_SENT_TAG, j);

      if (status.error != ILIB_SUCCESS ||
        status.size != sizeof(j) )
      {
        ilib_die("Failed ilib_msg_receive of buffer ready");
      }

      if (j != i)
      {
        ilib_die( "Receiver got message %d, expected message %d\n", j,
i);
      }

#ifdef DEBUG
      fprintf(stderr, "run_receiver: rank=%d, starting DMA of message
%d\n",
            my_rank, i);
      fflush(stderr);
#endif

      if (DMA_START(local_buf[slot], src_blocks[slot],
BUFFERSIZE(blocksize), requests[slot])
        < 0)
      {
        ilib_die("Failed 1st DMA.");
      }

#ifdef MEMCPY
      memcpy(sink_blocks[prevSlot], src_blocks[prevSlot],
BUFFERSIZE(blocksize));
#else
      fftwf_execute_dft_r2c(plan, src_blocks[prevSlot], (fftwf_complex
*)sink_blocks[prevSlot]);
#endif


      SEND_MESSAGE(SINK_RANK, MESSAGE_3_SENT_TAG, prev);

#ifdef DEBUG
      fprintf(stderr, "run_receiver: rank = %d, sink_block[0] = %f\n",
            my_rank, sink_blocks[prevSlot][0]);
      fflush(stderr);
#endif

      if (ilib_wait(&requests[slot], &status) < 0)
      {
        ilib_die("Failed from ilib_wait().");
      }
    }
  i = i--;

#ifdef DEBUG
  fprintf(stderr, "run_received: rank=%d, expect %d, src__blocks[0] =
%f\n,",
        my_rank, i, src_blocks[i % 2][0]);
  fflush(stderr);
#endif

#ifdef MEMCPY
  memcpy(sink_blocks[i %2 ], src_blocks[i %2], BUFFERSIZE(blocksize));
#else
  fftwf_execute_dft_r2c(plan, src_blocks[i%2], (fftwf_complex
*)sink_blocks[i%2]);
#endif

#ifdef DEBUG
  fprintf(stderr, "run_receiver: rank=%d, expect %d, src__blocks[slot]
= %f\n,",
        my_rank, i, src_blocks[slot][0]);
  fflush(stderr);

  fprintf(stderr, "run_receiver: rank=%d, sending msg %d ready\n",
my_rank, i);
  fflush(stderr);
#endif

  SEND_MESSAGE(SINK_RANK, MESSAGE_3_SENT_TAG, i);

  if (status.error != ILIB_SUCCESS )
    {
      ilib_die("run_receiver: fail on end of run on receiver %d\n",
my_rank);
    }

#ifdef PRINT_RECEIVER_OUT
  end_ctr = get_cycle_count();
  cycles = end_ctr - start_ctr;
  printf("run_receiver: rank=%d, Block size is %d bytes\n", my_rank,
BUFFERSIZE(blocksize));
  printf("run_receiver: rank=%d, last block expect %d : block[0] = %f,
block[%d] = %f\n",
       my_rank, i, local_buf[(i) % 2][0], i, local_buf[(i) %
2][blocksize-1]);
  long long blocksPerSec = 400000000L / (cycles/iters);
  printf("run_receiver: rank=%d, Cycles per transfer = %lld\n",
my_rank, cycles/iters);
  printf("run_receiver: rank=%d, Transfer Rate = %lld bytes/sec\n",
       my_rank, blocksPerSec * BUFFERSIZE(blocksize));
#endif
}

int main(void)
{
  ilib_init();
  const int my_rank = ilib_group_rank(ILIB_GROUP_SIBLINGS);

#ifdef DEBUG
  FILE *rsinkerr = fopen("run_sink_errors", "w");
  fprintf(stderr, "run_receiver: rank=%d, waiting for param list\n",
my_rank);
  fflush(stderr);
#endif

  if (my_rank == SINK_RANK || my_rank == SEND_RANK)
    {
      ilib_die("run_receiver: got rank = %d, wrong for receiver\n",
my_rank);
    }
  run_receiver();

#ifdef MEMCPY
  printf("Run Receiver used MEMCPY\n");
#else
  //  printf("Run Receiver used fftw\n");
#endif

  ilib_finish();
  return 0;
}
```

6. run_sink.c

```
/* Written by George W. Dinolt, Naval Postgraduate School, Monterey, CA
   Last Change Rev: 120, Last Changed Date: Fri, 22 Nov. 2013 */

/*
   run_sink: Program to receive all the computed fft's.


   The Flow is:

   RECEIVE BROADCAST message from run_sender with paramters (buffer
size).

   Setup local buffers

   receiver buffer addresses from run_senders buffers

   LOOP (iterations):
     for each receiver, receiver message that message is ready
                       dma message from receiver

  SEND_MESSAGE that we are done.


  Note that we are not trying to do multiple dma's symultaneously.
  since we are not doing processing here.  This would probably need to
  be fixed.
```

```
   See simpledma.h for descriptions of the macros:

   SEND_MESSAGE
   RECEIVE MESSAGE
   DMA_START
   BROADCAST_MESSAGE

define PRINT_SINK_OUT to test output
define DEBUG for debugging

*/

#include "simpledma.h"

//#define PRINT_SINK_OUT
//#define DEBUG

void run_sink(
#ifndef DEBUG
            void
#else
            FILE *rsinkerr
#endif
            )
{
  int  i;
  int  j;
  int  k;
  int  slot;
  long long end_ctr;

#ifdef PRINT_SINK_OUT
  long long start_ctr, cycles;
#endif

  int my_rank = ilib_group_rank(ILIB_GROUP_SIBLINGS);

  if (my_rank != SINK_RANK)
    {
      ilib_die("Got rank %d != SINK_RANK = %d\n", my_rank, SINK_RANK);
    }
  params_t *p;

#ifdef DEBUG
  fprintf(rsinkerr, "run_sink: rank=%d, waiting for param list\n",
my_rank);
  fflush(rsinkerr);
#endif

  ilibStatus status;
  // Receive paramaters from sender via broadcast
  BROADCAST_MESSAGE(SEND_RANK, p);
  if (status.error != ILIB_SUCCESS ||
      status.size != sizeof(p)  )
  {
    ilib_die("Failed receive of param broadcast");
  }
```

```
#ifdef DEBUG
  fprintf(rsinkerr, "run_sink: got parameter address %x\n", p);
  fflush(rsinkerr);
#endif

  const int blocksize = p->blocksize;
  const int iters = p->iters;
  const int nreceivers = p->nreceivers;

  float  *buffer;     /* source  buffer address */
  float  local_buf[2][blocksize];    /* destinatin buffers */
  float *src_blocks[nreceivers][2]; /* these are set to point to the
source buffers */
  ilibRequest requests[2];

  for ( k = 0; k < nreceivers; k++)
    {

#ifdef DEBUG
      fprintf(rsinkerr,"run_sink: waiting buffer address from receiver
%d\n", k);
      fflush(rsinkerr);
#endif
      RECEIVE_MESSAGE(k+FIRST_RECEIVER, MESSAGE_3_TAG, buffer);
      if (status.error != ILIB_SUCCESS ||
        status.size != sizeof(float)  )
      {
        ilib_die("run_sink: Failed to receive buffer addr from receiver
%d", k);
      }

#ifdef DEBUG
      fprintf(rsinkerr,"run_sink: GOT buffer address from receiver
%d\n", k);
      fflush(rsinkerr);
#endif

      src_blocks[k][0] = buffer;
      src_blocks[k][1] = buffer + blocksize;

#ifdef DEBUG
      fprintf(rsinkerr, "run_sink: SET source blocsk address for
receiver %d\n", k);
      fflush(rsinkerr);
#endif

    }

#ifdef DEBUG
  fprintf(rsinkerr, "run_sink: set buffers ok\n");
  fflush(rsinkerr);
#endif

#ifdef PRINT_SINK_OUT
  start_ctr = get_cycle_count();
#endif
```

```
  for (i = 0; i < iters; i++)
    {
      slot = i % 2;
      for (k = 0; k < nreceivers; k++)
      {
#ifdef DEBUG
        fprintf(rsinkerr, "run_sink: awating %d from receiver %d\n",
              i, k);
        fflush(rsinkerr);
#endif
        RECEIVE_MESSAGE(k+FIRST_RECEIVER, MESSAGE_3_SENT_TAG,j);
        if (status.error != ILIB_SUCCESS ||
            status.size != sizeof(j) )
          {
            ilib_die("run_sink: Failed ilib_msg_receive of buffer ready
from receiver %d",
                     k);
          }

        if (j != i)
          {
            ilib_die("Sink got message %d from receiver %d, expected
message %d\n",
                     j, k, i);
          }

        if (DMA_START(local_buf[slot], src_blocks[k][slot],
                    BUFFERSIZE(blocksize), requests[slot]) < 0)
          {
            ilib_die("run_sink, Failed 1st DMA from receiver %d.", k);
          }
        if (ilib_wait(&requests[slot], &status) < 0)
          {
            ilib_die("Failed from ilib_wait().");
          }
#ifdef DEBUG
        fprintf(rsinkerr, "run_sink: Received buffer %d from receiver
%d, block[0] = %f\n",
              j, k, src_blocks[k][slot][0]);
        fflush(rsinkerr);
#endif
      }
    }

  end_ctr = get_cycle_count();
  SEND_MESSAGE(SEND_RANK, MESSAGE_LAST_RECEIVED, end_ctr);

#ifdef PRINT_SINK_OUT
  cycles = end_ctr - start_ctr;
  printf("Sink Block size is %d bytes\n", blocksize * sizeof(float));
  for (k = 0; k < nreceivers; k++)
    {
      printf("run_sink, receiver %d, expected  %d : block[0] = %f,
block[%d] = %f\n",
           k, (i-1), local_buf[slot][0], i-1,
local_buf[slot][blocksize-1]);
```

```
    }
  printf("run_sink: iters=%d, nreceivers=%d, BUFFERSIZE=%d\n",
       iters, nreceivers, BUFFERSIZE(blocksize));
  printf("run_sink: cycles = %lld\n", cycles);
  long long cyclesPerBlock = cycles/(iters * nreceivers);
  printf("run_sink: Cycles per transfer = %lld\n", cyclesPerBlock);
  printf("Sink Transfer Rate = %lld bytes/sec\n",
       ((long long)iters * (long long)nreceivers *
        (long long)BUFFERSIZE(blocksize) * 400000000L)/cycles);
#endif
}

int main(void)
{
  ilib_init();
  const int my_rank = ilib_group_rank(ILIB_GROUP_SIBLINGS);

#ifdef DEBUG
  FILE *rsinkerr = fopen("run_sink_errors", "w");
  fprintf(rsinkerr, "run_sink: starting process with rank %d\n",
        my_rank);
  fflush(rsinkerr);
#endif

  if (my_rank != SINK_RANK)
    {
      ilib_die("run_sink: got rank = %d, wrong for sink\n", my_rank);
    }
  run_sink(
#ifdef DEBUG
  rsinkerr
#endif
    );

  ilib_finish();
  return 0;
}
```

7. start3dma.c

```
/* Written by George W. Dinolt, Naval Postgraduate School, Monterey, CA
   Last Change Rev: 120, Last Changed Date: Fri, 22 Nov. 2013 */

#include "simpledma.h"


#define USAGE "USAGE: start3dma blocksize iterations"
#define UBLOCK(b)  b == 32 || b == 64 || b == 128 || b == 256 ||\
    b == 512 || b == 1024 || b == 2048 || 4096

args_t *parseArgs(int argc, char *argv[])
{
  args_t * a = (args_t *)tmc_cmem_memalign((size_t)64, sizeof(args_t));

  if (argc != 3 )
    {
      ilib_die("simple3dma: wrong number of args, got %d\n\t%s",
             argc, USAGE);
    }
  a->blocksize = atoi(argv[1]);
  if ( ! (UBLOCK(a->blocksize)) )
    {
      ilib_die("simple3dma: blocksize=%d not power of 2\n\t%s", a-
>blocksize, USAGE);
    }
  a->iters = atoi(argv[2]);
  if ( a->iters <= 0 )
    {
      ilib_die("start3dma: terations=%s must be positive\n%s",
             argv[2], USAGE);
    }
  a->nreceivers = 1;  // fixed size
  return a;
}

int main(int argc, char *argv[])
{
  ilib_init();
  args_t *a = parseArgs(argc, argv);

  ilibProcParam pparams[3]; // send, sink, receiver

  memset(pparams, 0, sizeof(pparams));

  pparams[0].num_procs = 1;
  pparams[0].binary_name = "run_sender";
  pparams[0].init_block = a;
  pparams[0].init_size = sizeof(args_t);
  pparams[0].tiles.x = 1;
  pparams[0].tiles.y = 3;
  pparams[0].tiles.width = 1;
  pparams[0].tiles.height = 1;
  pparams[1].num_procs = 1;
  pparams[1].binary_name = "run_sink";
  pparams[1].tiles.x = 3;
  pparams[1].tiles.y = 3;
  pparams[1].tiles.width = 1;
  pparams[1].tiles.height = 1;
  pparams[2].num_procs = 1;
  pparams[2].binary_name = "run_receiver";
  pparams[2].tiles.x = 2;
  pparams[2].tiles.y = 3;
  pparams[2].tiles.width = 1;
  pparams[2].tiles.height = 1;
  ilibGroup spawned_procs;
  if (ilib_proc_spawn(3, pparams, &spawned_procs) != ILIB_SUCCESS)
    {
      ilib_die("Failed to spawn.");
    }
  ilib_finish();
  return 0;
                                  }
```

APPENDIX B.

Data from the experiments is tabulated in the following text files. The format for the data is illustrated below:

`nreceivers` is the number of parallel FFT tiles, *p*.

`cycles` is the number of cycles, $\overline{C}(N, p)$.

`Transfer Rate` is the net sample rate of the pipeline FP FFT.

1. Data for *N* = 128

```
run_sender: iters=100000, nreceivers=1, BUFFERSIZE=512
run_sender: cycles = 2035656968
run_sender: Cycles per block transfer = 20356
run_sender:  Transfer Rate = 2200763 samples/sec
-------
run_sender: iters=100000, nreceivers=2, BUFFERSIZE=512
run_sender: cycles = 2199992035
run_sender: Cycles per block transfer = 10999
run_sender:  Transfer Rate = 4072742 samples/sec
-------
run_sender: iters=100000, nreceivers=3, BUFFERSIZE=512
run_sender: cycles = 2385886892
run_sender: Cycles per block transfer = 7952
run_sender:  Transfer Rate = 5633125 samples/sec
-------
run_sender: iters=100000, nreceivers=4, BUFFERSIZE=512
run_sender: cycles = 2287643473
run_sender: Cycles per block transfer = 5719
run_sender:  Transfer Rate = 7833388 samples/sec
-------
run_sender: iters=100000, nreceivers=5, BUFFERSIZE=512
run_sender: cycles = 2566313364
run_sender: Cycles per block transfer = 5132
run_sender:  Transfer Rate = 8728474 samples/sec
-------
run_sender: iters=100000, nreceivers=6, BUFFERSIZE=512
run_sender: cycles = 3107507609
run_sender: Cycles per block transfer = 5179
run_sender:  Transfer Rate = 8650019 samples/sec
-------
run_sender: iters=100000, nreceivers=7, BUFFERSIZE=512
run_sender: cycles = 3254658571
run_sender: Cycles per block transfer = 4649
run_sender:  Transfer Rate = 9635419 samples/sec
```

```
-------
run_sender: iters=100000, nreceivers=8, BUFFERSIZE=512
run_sender: cycles = 3729559390
run_sender: Cycles per block transfer = 4661
run_sender:  Transfer Rate = 9609714 samples/sec
-------
run_sender: iters=100000, nreceivers=9, BUFFERSIZE=512
run_sender: cycles = 4042357031
run_sender: Cycles per block transfer = 4491
run_sender:  Transfer Rate = 9974378 samples/sec
-------
run_sender: iters=100000, nreceivers=10, BUFFERSIZE=512
run_sender: cycles = 4545065821
run_sender: Cycles per block transfer = 4545
run_sender:  Transfer Rate = 9856842 samples/sec
-------
run_sender: iters=100000, nreceivers=11, BUFFERSIZE=512
run_sender: cycles = 4931927446
run_sender: Cycles per block transfer = 4483
run_sender:  Transfer Rate = 9992036 samples/sec
-------
run_sender: iters=100000, nreceivers=12, BUFFERSIZE=512
run_sender: cycles = 5401540548
run_sender: Cycles per block transfer = 4501
run_sender:  Transfer Rate = 9952716 samples/sec
-------
run_sender: iters=100000, nreceivers=13, BUFFERSIZE=512
run_sender: cycles = 5799887319
run_sender: Cycles per block transfer = 4461
run_sender:  Transfer Rate = 10041574 samples/sec
-------
run_sender: iters=100000, nreceivers=14, BUFFERSIZE=512
run_sender: cycles = 6384539279
run_sender: Cycles per block transfer = 4560
run_sender:  Transfer Rate = 9823731 samples/sec
-------
run_sender: iters=100000, nreceivers=15, BUFFERSIZE=512
run_sender: cycles = 6702693130
run_sender: Cycles per block transfer = 4468
run_sender:  Transfer Rate = 10025820 samples/sec
-------
run_sender: iters=100000, nreceivers=16, BUFFERSIZE=512
run_sender: cycles = 7217814757
run_sender: Cycles per block transfer = 4511
run_sender:  Transfer Rate = 9930983 samples/sec
-------
run_sender: iters=100000, nreceivers=17, BUFFERSIZE=512
```

```
run_sender: cycles = 7310522304
run_sender: Cycles per block transfer = 4300
run_sender:  Transfer Rate = 10417860 samples/sec
-------
run_sender: iters=100000, nreceivers=18, BUFFERSIZE=512
run_sender: cycles = 7787227030
run_sender: Cycles per block transfer = 4326
run_sender:  Transfer Rate = 10355419 samples/sec
-------
run_sender: iters=100000, nreceivers=19, BUFFERSIZE=512
run_sender: cycles = 8376538489
run_sender: Cycles per block transfer = 4408
run_sender:  Transfer Rate = 10161715 samples/sec
-------
run_sender: iters=100000, nreceivers=20, BUFFERSIZE=512
run_sender: cycles = 8921883984
run_sender: Cycles per block transfer = 4460
run_sender:  Transfer Rate = 10042721 samples/sec
-------
```

2. Data for *N* = 256

```
run_sender: iters=100000, nreceivers=1, BUFFERSIZE=1024
run_sender: cycles = 2791582107
run_sender: Cycles per block transfer = 27915
run_sender:  Transfer Rate = 3209649 samples/sec
-------
run_sender: iters=100000, nreceivers=2, BUFFERSIZE=1024
run_sender: cycles = 3098993319
run_sender: Cycles per block transfer = 15494
run_sender:  Transfer Rate = 5782522 samples/sec
-------
run_sender: iters=100000, nreceivers=3, BUFFERSIZE=1024
run_sender: cycles = 3181487439
run_sender: Cycles per block transfer = 10604
run_sender:  Transfer Rate = 8448878 samples/sec
-------
run_sender: iters=100000, nreceivers=4, BUFFERSIZE=1024
run_sender: cycles = 3141524320
run_sender: Cycles per block transfer = 7853
run_sender:  Transfer Rate = 11408474 samples/sec
-------
run_sender: iters=100000, nreceivers=5, BUFFERSIZE=1024
run_sender: cycles = 3385977912
run_sender: Cycles per block transfer = 6771
run_sender:  Transfer Rate = 13231037 samples/sec
-------
run_sender: iters=100000, nreceivers=6, BUFFERSIZE=1024
```

```
run_sender: cycles = 3500426890
run_sender: Cycles per block transfer = 5834
run_sender:  Transfer Rate = 15358126 samples/sec
-------
run_sender: iters=100000, nreceivers=7, BUFFERSIZE=1024
run_sender: cycles = 3387679430
run_sender: Cycles per block transfer = 4839
run_sender:  Transfer Rate = 18514148 samples/sec
-------
run_sender: iters=100000, nreceivers=8, BUFFERSIZE=1024
run_sender: cycles = 3814325558
run_sender: Cycles per block transfer = 4767
run_sender:  Transfer Rate = 18792313 samples/sec
-------
run_sender: iters=100000, nreceivers=9, BUFFERSIZE=1024
run_sender: cycles = 4190221692
run_sender: Cycles per block transfer = 4655
run_sender:  Transfer Rate = 19244805 samples/sec
-------
run_sender: iters=100000, nreceivers=10, BUFFERSIZE=1024
run_sender: cycles = 4610221058
run_sender: Cycles per block transfer = 4610
run_sender:  Transfer Rate = 19435076 samples/sec
-------
run_sender: iters=100000, nreceivers=11, BUFFERSIZE=1024
run_sender: cycles = 5057583597
run_sender: Cycles per block transfer = 4597
run_sender:  Transfer Rate = 19487567 samples/sec
-------
run_sender: iters=100000, nreceivers=12, BUFFERSIZE=1024
run_sender: cycles = 5497382446
run_sender: Cycles per block transfer = 4581
run_sender:  Transfer Rate = 19558399 samples/sec
-------
run_sender: iters=100000, nreceivers=13, BUFFERSIZE=1024
run_sender: cycles = 5949133240
run_sender: Cycles per block transfer = 4576
run_sender:  Transfer Rate = 19579322 samples/sec
-------
run_sender: iters=100000, nreceivers=14, BUFFERSIZE=1024
run_sender: cycles = 6330117808
run_sender: Cycles per block transfer = 4521
run_sender:  Transfer Rate = 19816376 samples/sec
-------
run_sender: iters=100000, nreceivers=15, BUFFERSIZE=1024
run_sender: cycles = 6842435283
run_sender: Cycles per block transfer = 4561
```

```
run_sender:  Transfer Rate = 19642129 samples/sec
-------
run_sender: iters=100000, nreceivers=16, BUFFERSIZE=1024
run_sender: cycles = 7447425160
run_sender: Cycles per block transfer = 4654
run_sender:  Transfer Rate = 19249605 samples/sec
-------
run_sender: iters=100000, nreceivers=17, BUFFERSIZE=1024
run_sender: cycles = 7703084873
run_sender: Cycles per block transfer = 4531
run_sender:  Transfer Rate = 19773896 samples/sec
-------
run_sender: iters=100000, nreceivers=18, BUFFERSIZE=1024
run_sender: cycles = 8176055917
run_sender: Cycles per block transfer = 4542
run_sender:  Transfer Rate = 19725892 samples/sec
-------
run_sender: iters=100000, nreceivers=19, BUFFERSIZE=1024
run_sender: cycles = 8684910868
run_sender: Cycles per block transfer = 4571
run_sender:  Transfer Rate = 19601813 samples/sec
-------
run_sender: iters=100000, nreceivers=20, BUFFERSIZE=1024
run_sender: cycles = 9243195020
run_sender: Cycles per block transfer = 4621
run_sender:  Transfer Rate = 19387235 samples/sec
-------
```

3. Data for *N* = 512

```
run_sender: iters=100000, nreceivers=1, BUFFERSIZE=2048
run_sender: cycles = 4978489886
run_sender: Cycles per block transfer = 49784
run_sender:  Transfer Rate = 3599485 samples/sec
-------
run_sender: iters=100000, nreceivers=2, BUFFERSIZE=2048
run_sender: cycles = 5418243924
run_sender: Cycles per block transfer = 27091
run_sender:  Transfer Rate = 6614689 samples/sec
-------
run_sender: iters=100000, nreceivers=3, BUFFERSIZE=2048
run_sender: cycles = 5920562990
run_sender: Cycles per block transfer = 19735
run_sender:  Transfer Rate = 9080217 samples/sec
-------
run_sender: iters=100000, nreceivers=4, BUFFERSIZE=2048
run_sender: cycles = 6454451171
run_sender: Cycles per block transfer = 16136
```

```
run_sender:  Transfer Rate = 11105514 samples/sec
-------
run_sender: iters=100000, nreceivers=5, BUFFERSIZE=2048
run_sender: cycles = 7026632496
run_sender: Cycles per block transfer = 14053
run_sender:  Transfer Rate = 12751485 samples/sec
-------
run_sender: iters=100000, nreceivers=6, BUFFERSIZE=2048
run_sender: cycles = 7819453536
run_sender: Cycles per block transfer = 13032
run_sender:  Transfer Rate = 13750321 samples/sec
-------
run_sender: iters=100000, nreceivers=7, BUFFERSIZE=2048
run_sender: cycles = 5847600257
run_sender: Cycles per block transfer = 8353
run_sender:  Transfer Rate = 21451534 samples/sec
-------
run_sender: iters=100000, nreceivers=8, BUFFERSIZE=2048
run_sender: cycles = 6946443559
run_sender: Cycles per block transfer = 8683
run_sender:  Transfer Rate = 20637898 samples/sec
-------
run_sender: iters=100000, nreceivers=9, BUFFERSIZE=2048
run_sender: cycles = 8156943181
run_sender: Cycles per block transfer = 9063
run_sender:  Transfer Rate = 19772112 samples/sec
-------
run_sender: iters=100000, nreceivers=10, BUFFERSIZE=2048
run_sender: cycles = 9334719083
run_sender: Cycles per block transfer = 9334
run_sender:  Transfer Rate = 19197149 samples/sec
-------
run_sender: iters=100000, nreceivers=11, BUFFERSIZE=2048
run_sender: cycles = 10201284233
run_sender: Cycles per block transfer = 9273
run_sender:  Transfer Rate = 19323057 samples/sec
-------
run_sender: iters=100000, nreceivers=12, BUFFERSIZE=2048
run_sender: cycles = 8381768798
run_sender: Cycles per block transfer = 6984
run_sender:  Transfer Rate = 25655682 samples/sec
-------
run_sender: iters=100000, nreceivers=13, BUFFERSIZE=2048
run_sender: cycles = 9330323210
run_sender: Cycles per block transfer = 7177
run_sender:  Transfer Rate = 24968052 samples/sec
-------
```

```
run_sender: iters=100000, nreceivers=14, BUFFERSIZE=2048
run_sender: cycles = 9984176673
run_sender: Cycles per block transfer = 7131
run_sender:  Transfer Rate = 25127760 samples/sec
-------
run_sender: iters=100000, nreceivers=15, BUFFERSIZE=2048
run_sender: cycles = 10672738943
run_sender: Cycles per block transfer = 7115
run_sender:  Transfer Rate = 25185662 samples/sec
-------
run_sender: iters=100000, nreceivers=16, BUFFERSIZE=2048
run_sender: cycles = 11346573180
run_sender: Cycles per block transfer = 7091
run_sender:  Transfer Rate = 25269303 samples/sec
-------
run_sender: iters=100000, nreceivers=17, BUFFERSIZE=2048
run_sender: cycles = 11737141434
run_sender: Cycles per block transfer = 6904
run_sender:  Transfer Rate = 25955212 samples/sec
-------
run_sender: iters=100000, nreceivers=18, BUFFERSIZE=2048
run_sender: cycles = 12770581530
run_sender: Cycles per block transfer = 7094
run_sender:  Transfer Rate = 25258051 samples/sec
-------
run_sender: iters=100000, nreceivers=19, BUFFERSIZE=2048
run_sender: cycles = 13412111301
run_sender: Cycles per block transfer = 7059
run_sender:  Transfer Rate = 25386010 samples/sec
-------
run_sender: iters=100000, nreceivers=20, BUFFERSIZE=2048
run_sender: cycles = 14639811187
run_sender: Cycles per block transfer = 7319
run_sender:  Transfer Rate = 24481190 samples/sec
-------
```

4. Data for *N* = 1024

```
run_sender: iters=100000, nreceivers=1, BUFFERSIZE=4096
run_sender: cycles = 9903822734
run_sender: Cycles per block transfer = 99038
run_sender:  Transfer Rate = 3618804 samples/sec
-------
run_sender: iters=100000, nreceivers=2, BUFFERSIZE=4096
run_sender: cycles = 10839464034
run_sender: Cycles per block transfer = 54197
run_sender:  Transfer Rate = 6612873 samples/sec
-------
```

```
run_sender: iters=100000, nreceivers=3, BUFFERSIZE=4096
run_sender: cycles = 12168398157
run_sender: Cycles per block transfer = 40561
run_sender:  Transfer Rate = 8836002 samples/sec
-------
run_sender: iters=100000, nreceivers=4, BUFFERSIZE=4096
run_sender: cycles = 13697124748
run_sender: Cycles per block transfer = 34242
run_sender:  Transfer Rate = 10466430 samples/sec
-------
run_sender: iters=100000, nreceivers=5, BUFFERSIZE=4096
run_sender: cycles = 15811185628
run_sender: Cycles per block transfer = 31622
run_sender:  Transfer Rate = 11333748 samples/sec
-------
run_sender: iters=100000, nreceivers=6, BUFFERSIZE=4096
run_sender: cycles = 17417132283
run_sender: Cycles per block transfer = 29028
run_sender:  Transfer Rate = 12346464 samples/sec
-------
run_sender: iters=100000, nreceivers=7, BUFFERSIZE=4096
run_sender: cycles = 15139625774
run_sender: Cycles per block transfer = 21628
run_sender:  Transfer Rate = 16571083 samples/sec
-------
run_sender: iters=100000, nreceivers=8, BUFFERSIZE=4096
run_sender: cycles = 12741296888
run_sender: Cycles per block transfer = 15926
run_sender:  Transfer Rate = 22503203 samples/sec
-------
run_sender: iters=100000, nreceivers=9, BUFFERSIZE=4096
run_sender: cycles = 15022229028
run_sender: Cycles per block transfer = 16691
run_sender:  Transfer Rate = 21472179 samples/sec
-------
run_sender: iters=100000, nreceivers=10, BUFFERSIZE=4096
run_sender: cycles = 14879141915
run_sender: Cycles per block transfer = 14879
run_sender:  Transfer Rate = 24087410 samples/sec
-------
run_sender: iters=100000, nreceivers=11, BUFFERSIZE=4096
run_sender: cycles = 14892368820
run_sender: Cycles per block transfer = 13538
run_sender:  Transfer Rate = 26472618 samples/sec
-------
run_sender: iters=100000, nreceivers=12, BUFFERSIZE=4096
run_sender: cycles = 15735538711
```

```
run_sender: Cycles per block transfer = 13112
run_sender:  Transfer Rate = 27331762 samples/sec
-------
run_sender: iters=100000, nreceivers=13, BUFFERSIZE=4096
run_sender: cycles = 20960949070
run_sender: Cycles per block transfer = 16123
run_sender:  Transfer Rate = 22228001 samples/sec
-------
run_sender: iters=100000, nreceivers=14, BUFFERSIZE=4096
run_sender: cycles = 22012817294
run_sender: Cycles per block transfer = 15723
run_sender:  Transfer Rate = 22793992 samples/sec
-------
run_sender: iters=100000, nreceivers=15, BUFFERSIZE=4096
run_sender: cycles = 22325582293
run_sender: Cycles per block transfer = 14883
run_sender:  Transfer Rate = 24079999 samples/sec
-------
run_sender: iters=100000, nreceivers=16, BUFFERSIZE=4096
run_sender: cycles = 21544904761
run_sender: Cycles per block transfer = 13465
run_sender:  Transfer Rate = 26616037 samples/sec
-------
run_sender: iters=100000, nreceivers=17, BUFFERSIZE=4096
run_sender: cycles = 24711916375
run_sender: Cycles per block transfer = 14536
run_sender:  Transfer Rate = 24655311 samples/sec
-------
run_sender: iters=100000, nreceivers=18, BUFFERSIZE=4096
run_sender: cycles = 24265874835
run_sender: Cycles per block transfer = 13481
run_sender:  Transfer Rate = 26585482 samples/sec
-------
run_sender: iters=100000, nreceivers=19, BUFFERSIZE=4096
run_sender: cycles = 25443068580
run_sender: Cycles per block transfer = 13391
run_sender:  Transfer Rate = 26764067 samples/sec
-------
run_sender: iters=100000, nreceivers=20, BUFFERSIZE=4096
run_sender: cycles = 29676664231
run_sender: Cycles per block transfer = 14838
run_sender:  Transfer Rate = 24153658 samples/sec
-------
```

5. Data for *N* = 2048

```
run_sender: iters=100000, nreceivers=1, BUFFERSIZE=8192
run_sender: cycles = 21957583650
```

```
run_sender: Cycles per block transfer = 219575
run_sender:  Transfer Rate = 3264475 samples/sec
-------
run_sender: iters=100000, nreceivers=2, BUFFERSIZE=8192
run_sender: cycles = 25786960848
run_sender: Cycles per block transfer = 128934
run_sender:  Transfer Rate = 5559398 samples/sec
-------
run_sender: iters=100000, nreceivers=3, BUFFERSIZE=8192
run_sender: cycles = 29117933559
run_sender: Cycles per block transfer = 97059
run_sender:  Transfer Rate = 7385139 samples/sec
-------
run_sender: iters=100000, nreceivers=4, BUFFERSIZE=8192
run_sender: cycles = 30139314757
run_sender: Cycles per block transfer = 75348
run_sender:  Transfer Rate = 9513155 samples/sec
-------
run_sender: iters=100000, nreceivers=5, BUFFERSIZE=8192
run_sender: cycles = 32404723357
run_sender: Cycles per block transfer = 64809
run_sender:  Transfer Rate = 11060116 samples/sec
-------
run_sender: iters=100000, nreceivers=6, BUFFERSIZE=8192
run_sender: cycles = 35749882192
run_sender: Cycles per block transfer = 59583
run_sender:  Transfer Rate = 12030249 samples/sec
-------
run_sender: iters=100000, nreceivers=7, BUFFERSIZE=8192
run_sender: cycles = 34799440259
run_sender: Cycles per block transfer = 49713
run_sender:  Transfer Rate = 14418622 samples/sec
-------
run_sender: iters=100000, nreceivers=8, BUFFERSIZE=8192
run_sender: cycles = 25518142889
run_sender: Cycles per block transfer = 31897
run_sender:  Transfer Rate = 22471854 samples/sec
-------
run_sender: iters=100000, nreceivers=9, BUFFERSIZE=8192
run_sender: cycles = 35749429667
run_sender: Cycles per block transfer = 39721
run_sender:  Transfer Rate = 18045602 samples/sec
-------
run_sender: iters=100000, nreceivers=10, BUFFERSIZE=8192
run_sender: cycles = 36182445261
run_sender: Cycles per block transfer = 36182
run_sender:  Transfer Rate = 19810711 samples/sec
```

```
-------
run_sender: iters=100000, nreceivers=11, BUFFERSIZE=8192
run_sender: cycles = 34904862567
run_sender: Cycles per block transfer = 31731
run_sender:  Transfer Rate = 22589402 samples/sec
-------
run_sender: iters=100000, nreceivers=12, BUFFERSIZE=8192
run_sender: cycles = 38809087133
run_sender: Cycles per block transfer = 32340
run_sender:  Transfer Rate = 22163881 samples/sec
-------
run_sender: iters=100000, nreceivers=13, BUFFERSIZE=8192
run_sender: cycles = 39326853112
run_sender: Cycles per block transfer = 30251
run_sender:  Transfer Rate = 23694751 samples/sec
-------
run_sender: iters=100000, nreceivers=14, BUFFERSIZE=8192
run_sender: cycles = 42482567546
run_sender: Cycles per block transfer = 30344
run_sender:  Transfer Rate = 23621924 samples/sec
-------
run_sender: iters=100000, nreceivers=15, BUFFERSIZE=8192
run_sender: cycles = 41767561157
run_sender: Cycles per block transfer = 27845
run_sender:  Transfer Rate = 25742465 samples/sec
-------
run_sender: iters=100000, nreceivers=16, BUFFERSIZE=8192
run_sender: cycles = 45359091641
run_sender: Cycles per block transfer = 28349
run_sender:  Transfer Rate = 25284456 samples/sec
-------
run_sender: iters=100000, nreceivers=17, BUFFERSIZE=8192
run_sender: cycles = 47660219939
run_sender: Cycles per block transfer = 28035
run_sender:  Transfer Rate = 25567653 samples/sec
-------
run_sender: iters=100000, nreceivers=18, BUFFERSIZE=8192
run_sender: cycles = 48612930279
run_sender: Cycles per block transfer = 27007
run_sender:  Transfer Rate = 26541086 samples/sec
-------
run_sender: iters=100000, nreceivers=19, BUFFERSIZE=8192
run_sender: cycles = 50663798563
run_sender: Cycles per block transfer = 26665
run_sender:  Transfer Rate = 26881521 samples/sec
-------
run_sender: iters=100000, nreceivers=20, BUFFERSIZE=8192
```

```
run_sender: cycles = 51987647456
run_sender: Cycles per block transfer = 25993
run_sender:  Transfer Rate = 27575781 samples/sec
-------
```

LIST OF REFERENCES

[1] Herschel Loomis, et al., "Reliable Architectures for High-Performance Space-Based Software Defined Radios (U)," Naval Postgraduate School, Monterey, CA, 2011.

[2] D. Summers, *Design and Construction of a Fault-tolerant Computer,* Monterey, CA: Naval Postgraduate School, 2000.

[3] Boeing Corporation, "Intellectual Property Summary," Seattle, 2010.

[4] "OPERA project Industry Day," Aerospace Corporation (J.P. Campbell Auditorium), 15049 Conference Center Drive, Chantilly, VA, 10/29-30/2010.

[5] Air Force Research Laboratory (AFRL) Program manager-creigh.gordon@kirtland.af.mil, "Air Force Research Laboratory (AFRL) Single Event Immune Reconfigurable FPGA (SIRF) program (Xilinx V5QV)," Sunnyvale, CA, 2011.

[6] Earl Fuller, Michael Caffrey, Anthony Salazar, Carl Carmichael, Joe Fabula, "Radiation Characterization and SEU Mitigation of the Virtex FPGA for Space-Based Reconfigurable Computing," in *Proceedings of NSREC2000*.

[7] Capt. Charles A. Hulme, USMC, Dr. Herschel H. Loomis, Dr. Alan A. Ross, and 1Lt. Rong Yuan, Taiwan Air Force, "Configurable Fault-Tolerant Processor (CFTP) for Spacecraft Onboard Processing," in *Proceedings 2004 IEEE Aerospace Conference*, Big Sky, Montana, March 2004.

[8] Wright, Durke, LT USN, "Field Programmable Gate Array (FPGA) Based Software Defined Radio (SDR) Design," MSEE Thesis, Naval Postgraduate School, Monterey, CA, March 2009.

[9] Livingston, Jeremy, "A Field Programmable Gate Array Based Software Defined Radio Design for the Space Environment," MSEE Thesis, Naval Postgraduate School, Monterey, CA, December 2009.

[10] Raymond F. Bernstein, Jr., "A Pipelined Vector Processor and Memory Architecture for Cyclostationary Processing," PhD Dissertation, Naval Postgraduate School, Monterey, CA, December 1995.

[11] LTJG Athanasios Gavros, Prof. Herschel H. Loomis, Jr., Prof. Alan A. Ross, "Reduced Precision Redundancy in a Radix-4 FFT Implementation on a Field Programmable Gate Array," in *Proceedings of the 2011 IEEE Aerospace Conference* , Big Sky, Montana, March 2011.

[12] Herschel Loomis and Alan Ross , "OPERA/MAESTRO Fault Tolerance Issues," Naval Postgraduate School Report, Monterey, CA, 31 July 2009.

[13] Humberd, Caleb J., LT USN, "A Compression Algorithm for Field Programmable Gate Arrays in the Space Environment," MSEE Thesis, Naval Postgraduate School, Monterey, CA, December 2011.

[14] Karandeep Singh, John Paul Walters, Joel Hestness, Jinwoo Suh, Craig M. Rogers,

Stephen P. Crago, "FFTW and Complex Ambiguity Function Performance on the Maestro Processor," in *Proceedings, 2011 IEEE Aerospace Conference*, Big Sky, Montana, March, 2011.

[15] "Maestro Software and Documentation IP Web Site," Maestro Project, [Online]. Available: https://opera.isi.edu/Software_IP. [Accessed 2013].

INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center
   Ft. Belvoir, Virginia

2. Dudley Knox Library
   Naval Postgraduate School
   Monterey, California

3. Research Sponsored Programs Office, Code 41
   Naval Postgraduate School
   Monterey, CA 93943

4. Maestro project wiki
   c/o Martha Bancroft
   marti@dragonsden.com

5. Dr. Stephen P. Crago
   University of Southern California / Information Sciences Institute
   3811 N. Fairfax, Suite 200
   Arlington, VA 22203
   crago@isi.edu

THIS PAGE INTENTIONALLY LEFT BLANK